

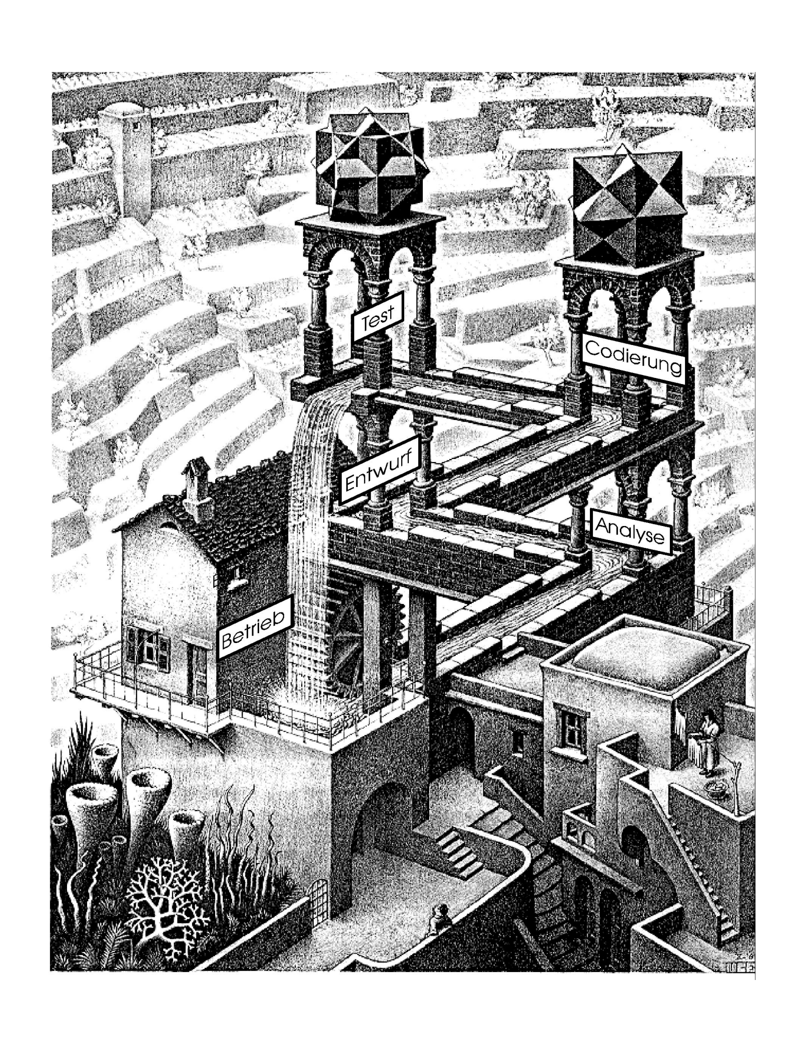


**Mathematisch-
Naturwissenschaftliche
Fakultät**

Skriptum softwaretechnik (Entwurf)

Herbert Klaeren

29. Juni 2015



Abbildungsnachweis: Titelbild: M. C. Escher / Volker Klaeren, Abb. 3, 5–7, 15–18, 22–23, 39: Volker Klaeren; Abb. 20, 40, 41–47: Frank Gerhardt.

Zahlreiche verbesserungsvorschläge von Markus Leypold und Mandeep Singh Multani wurden dankbar entgegengenommen. Zum abschnitt über objektorientierte softwareentwicklung hat Andreas Speck wertvolle beiträge geliefert.

Eine bemerkung zur rechtschreibung: Deutschland diskutiert permanent über die rechtschreibung, vermeidet dabei aber echte fortschritte. Dieses skriptum verwendet zum teil eine experimentelle orthographie nach den folgenden drei regeln:

1. Grossgeschrieben werden satzanfänge, eigennamen und kapitalisierte abkürzungen (CORBA, OMG, TCP, UML, . . .); alles andere wird kleingeschrieben.

2. Das „scharfe s" („Eszet") wird ausser in eigennamen überall durch „ss" ersetzt. Die Schweizer können's schliesslich auch so. Was es bedeuten mag, „schokolade in massen" zu geniessen, überlasse ich der phantasie der leserin.

3. Wörtliche zitate verwenden die an der fundstelle vorhandene schreibung.

Prof. Dr. H. Klaeren
Universität Tübingen
Wilhelm-Schickard-Institut
D–72076 Tübingen
+49–7071–29 75457
`klaeren@informatik.uni-tuebingen.de`

Inhaltsverzeichnis

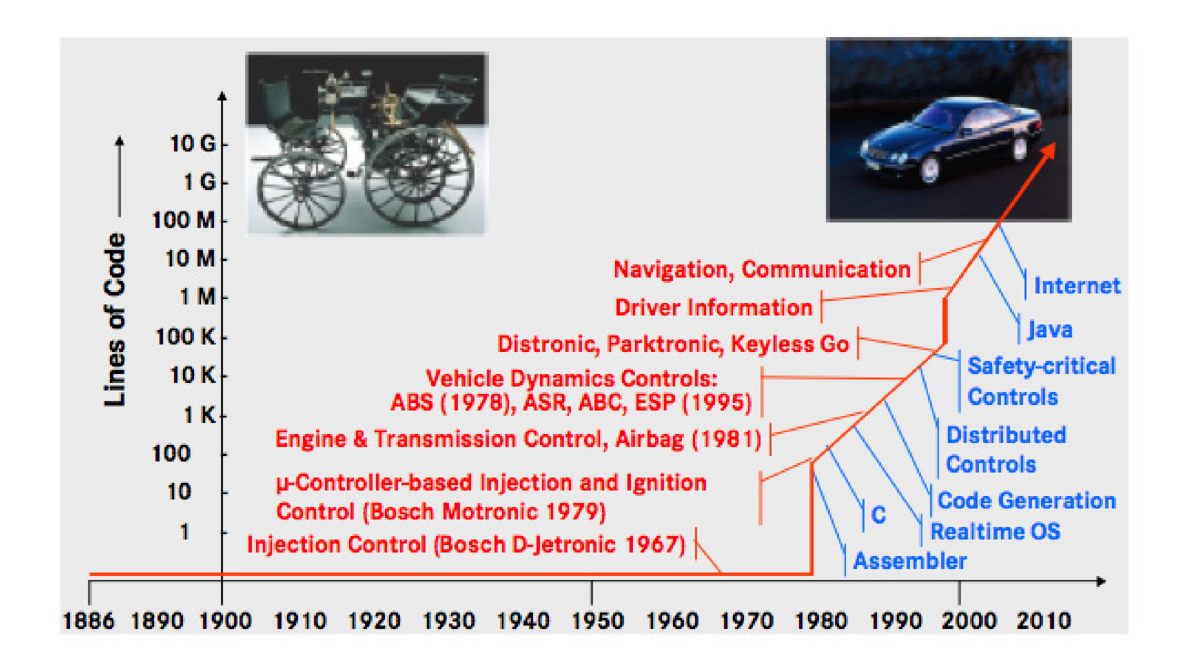


Abbildung 1: Software im PKW

1 Einführung: Was ist Softwaretechnik?

1.1 Motivation

Funktionierende software wird ständig wichtiger für das funktionieren der welt; immer mehr vorgänge finden computergesteuert statt und immer mehr von den leistungen komplexer systeme wird in deren softwareanteil verlagert. Die automobiltechnik bietet hierzu grossartige beispiele; siehe abb. 1 aus der publikation von Klein u. a. (2004), die auch deutlich macht, wozu die software im automobil verwendet wird[1]. Oft lässt sich ein unerwünschtes oder gar gefährliches fahrverhalten von autos durch eine änderung der software korrigieren (s. etwa den artikel aus der Südwestpresse in anhang E, S. 244).

Der trend zu softwarelösungen hat zum grossen teil mit den produktionskosten zu tun: jede noch so clever konstruierte elektronik verursacht trotzdem immer noch kosten bei ihrer replikation, während sich eine einmal erstellte software praktisch zum nulltarif vervielfachen lässt. Andererseits ist die herstellung der software nicht ganz billig: Die *International Technology Roadmap for Semiconductors* 2010 (`www.itrs.net`) berichtet, dass von den entwicklungskosten für eingebettete systeme 80% auf die entwicklung der software entfallen.

Von industriellen erzeugnissen wie waschmaschinen, autos, fernseher, sind

[1] Der begriff „Lines of Code“ wird in abschnitt 1.4 aufgegriffen.

wir es gewöhnt, dass die hersteller eine gewisse garantie für das korrekte funktionieren des produkts über einen gewissen zeitraum übernehmen. Garantiebestimmungen für software sagen dagegen in der regel nicht viel mehr aus als dass die CDs silbern, rund und mit maschinenlesbaren aufzeichnungen versehen sind. Das einzige recht, das dem kunden eingeräumt wird, ist das recht auf rückzahlung des kaufpreises.

Auch wenn nicht alle horrormeldungen über softwareprodukte ernstgenommen werden sollten (Glass 2005), so berichten die medien dennoch genüsslich von software-desastern; manche sprechen von „bananensoftware“ (sie reift beim kunden). Die Computerzeitung (Reiter 2008) schreibt:

> *Einer Studie von Corporate Quality beziffert die Schäden durch Softwarepannen allein in Deutschland auf bis zu 100 Milliarden Euro. Diese Zahl ergibt sich dann, wenn alle Auswirkungen mangelhafter IT-Strukturen mit eingerechnet werden. Alleine durch schlechte Software respektive Ineffektivität entgehen den Unternehmen jährlich über 20 Milliarden Euro, so die Expertenbefragung.*

Das Handelsblatt (Nr. 153, 12. August 1998, S. 37) lästert:

> *Software ist das fehlerhafteste Produkt überhaupt. Es ist schon erstaunlich, wie es manche Anwender mit der Geduld einer tibetanischen Bergziege hinnehmen, was ihnen da Programmierer zumuten. [. . .] Programme machen, was sie wollen, nur nicht das, was der Anwender gerne hätte. Das System stürzt ab, Daten gehen verloren. Selbst die Gerichte haben kein Einsehen mit dem geplagten Anwender; sie vertreten schlicht den Standpunkt, daß Software nun einmal fehlerhaft sei. Gewährleistung ist für die Softwarehersteller kein Thema.*

Die folgenden beispiele sind nur eine kleine auswahl; diese liste lässt sich praktisch endlos fortsetzen[2].

- In Warschau rollt ein Airbus über die landebahn hinaus und geht in flammen auf, weil die bordcomputer die auslösung der schubumkehr verweigern (Neumann 1995a, p. 46): das flugzeug gilt erst als gelandet, wenn alle drei fahrwerke belastet sind und die räder sich drehen. Eine korrektur dieses verhaltens führt später zu einem beinahe-unfall in Hamburg: Nachdem ein fahrwerk auf dem boden aufgesetzt hat, schaltet das flugzeug in den bodenmodus, wodurch unter anderem der querrudereinschlag begrenzt wird. Ein plötzlicher seitenwind schleudert das flugzeug auf die andere seite, die tragfläche berührt den boden, ohne dass die piloten etwas dagegen tun können (Süddeutsche Zeitung 2010; Traufetter 2010; Reuß 2010).

[2] Eine grosse zusammenstellung von computerfehlern (nicht nur im zusammenhang mit software) wird von P. G. Neumann (1995a) geführt.

- Im neuen flughafen Denver versagt die softwaresteuerung der gepäcktransportbänder derart, dass koffer zerrissen werden (Swartz 1996). Der flughafen wird mit 16-monatiger verspätung eröffnet bei einem verdienstausfall von rund einer million Dollar pro tag.
- Durch eine falsche software gerät die für 11 milliarden Mark entwickelte Ariane 5 ausser kontrolle und muss gesprengt werden (Jézéquel und Meyer 1997).
- Das *Toll Collect*-system für die LKW-maut auf deutschen autobahnen soll am 31. August 2003 in dienst gehen, erreicht aber erst am 1. Januar 2006 seine volle funktionalität. Die bundesregierung klagt gegen das konsortium wegen 1,6 milliarden Euro vertragsstrafe und 3,5 milliarden Euro einnahmeausfällen (Wikipedia 2008).
- Zu kurze kabelbäume beim bau des A380 verursachen bei Airbus schäden von 4,8 millionen Euro (Weiss 2008); ursache ist der einsatz zweier inkompatibler versionen des gleichen CAD-programms.

1.1.1 Ursachen

Es gibt mehrere gründe, warum software oft nicht die in sie gesetzten erwartungen erfüllt:

1. *Software ist meistens eingebettet in komplexe systeme, die nicht primär als computer dienen.* Wer bei dem wort „software“ an seinen PC und die mehr oder weniger nützlichen applikationen darauf denkt, denkt nicht weit genug: 98% der programmierbaren CPUs werden heute in „eingebetteten systemen“ verwendet (Broy und Pree 2003); d.h. rechner und ihre programme stellen nur komponenten in systemen dar, die sich „von aussen“ gar nicht als computer präsentieren (waschmaschinen, videorecorder, mobiltelefone, autos, ...)
2. *Die aufgabe der software in einem komplexen system ist schwer zu definieren, weil die interaktion des computers mit der umwelt unübersichtlich ist.*
3. *Das eigentliche programmieren ist nur ein (kleiner) teil der tätigkeit.* Für jedes softwareprojekt sind zahlreiche besprechungen notwendig, um erst einmal den *funktionsumfang* der software zu beschreiben, *zuständigkeiten* auf personen zu verteilen, die in unterschiedlichen *rollen* im projekt auftreten, *termine* festzulegen und den *projektfortschritt* zu kontrollieren etc.
4. *Softwarepakete haben gewaltige dimensionen und werden von grossen teams hergestellt.* Das hat gravierende auswirkungen auf den *prozess* der softwareerstellung.

5. *Die digitale welt impliziert viel komplexere abhängigkeiten als die analoge welt der stetigen funktionen.* Der begriff der stetigen funktion drückt gerade das prinzip aus, dass kleine änderungen kleine oder doch wenigstens verhältnismässige auswirkungen haben. So wird z.b. ein stab, auf den man nach und nach grössere biegekräfte ausübt, innerhalb eines gewissen bereichs seine form in genauer entsprechung zu der ausgeübten kraft verändern. Natürlich gibt es unstetigkeitsstellen, wenn z.b. die ausgeübte kraft dazu führt, dass der stab bricht; aber es gehört zum handwerk des ingenieurs, die zahlenmässig wenigen unstetigkeitsstellen zu erkennen und im normalen betrieb seiner konstruktion zu vermeiden. In der regel hat man es mit stetigen funktionen zu tun.

   In der softwaretechnik haben wir es demgegenüber von hause aus mit einer *digitalen welt* zu tun. Während in einem analogen modell die modellierende grösse ein *direktes abbild* der entsprechenden physikalischen grösse ist, wird in einer digitalen darstellung eine pysikalische grösse durch eine *symbolische beschreibung*, z.b. zahlen in einer gewissen notation, dargestellt. Der begriff der stetigen funktion spielt hier keine rolle, es herrscht vielmehr eine maximale unstetigkeit. Wenn man in einer 64-bit-binärzahl eine einzige stelle von 0 zu 1 ändert, entsteht nicht unbedingt eine „ganz ähnliche“, sondern unter umständen eine weit entfernte zahl. Selbst kleinste änderungen können also grösste auswirkungen haben, siehe auch abb. 2.

1.2 Ingenieurwissenschaft

Unsere vorlesung heisst *softwaretechnik*; anderswo heisst diese veranstaltung etwas vornehmer, weil englisch, *software engineering*. Laut lexikon wäre „softwaretechnik“ die präzise übersetzung von *software engineering*; allerdings wird die englische bezeichnung von vielen leuten auch in deutschen texten bevorzugt, weil sie deutlicher macht, dass die konstruktion von software eine *ingenieurtätigkeit* ist oder sein sollte. Die mit dem 1968 eingeführten begriff „software engineering“ verbundene absicht war in der tat, die qualität der programme zu verbessern, indem man ingenieursprinzipien auf ihre herstellung anwandte (s. anh. C). Bevor wir richtig beginnen, daher zunächst einige definitionen aus der literatur:

Software Engineering (nach dem IEEE Standard Glossary of Software Engineering Terminology , IEEE (1990)) *the systematic approach to the development, operation, maintenance, and retirement of software*

„der systematische ansatz zu entwicklung, betrieb, wartung und ausser-



Abbildung 2: Analog und digital

betriebnahme[3] von software“

Software (nach IEEE (1990)) *programs, procedures, rules, and any associated documentation pertaining to the operation of a computer system*

„Programme, verfahren und regeln sowie jegliche zugehörige dokumentation zum betrieb eines computersystems“

Sehr viel geben diese definitionen zugegebenermassen nicht her. Etwas mehr einsicht in die hinter dem begriff *software engineering* stehende gedankenwelt bekommen wir, wenn wir das wort *engineering* in einem guten englischen lexikon, z. B. der Encyclopædia Britannica (1999) nachschlagen. Dort wird zunächst eine Definition des *Engineers Council for Professional Development* aus den USA zitiert; derzufolge ist *engineering*

> *„die schöpferische anwendung wissenschaftlicher prinzipien auf entwurf und entwicklung von strukturen, maschinen, apparaten oder herstellungsprozessen oder arbeiten, wobei diese einzeln oder in kombination verwendet werden; oder dies alles zu konstruieren und zu betreiben in voller kenntnis seines entwurfs; oder dessen verhalten unter bestimmten betriebsbedingungen vorherzusagen; alles dies im hinblick auf eine gewünschte funktion, wirtschaftlichkeit des betriebs und sicherheit von leben und eigentum“*

Dies wird dann anschliessend in einem längeren artikel näher ausgeführt. Wir finden im obigen zitat drei wirklich spezifische wesensmerkmale der tätigkeit eines ingenieurs vor:

Kreative anwendung wissenschaftlicher grundlagen zur erbringung einer gewissen funktion Der ingenieur hat immer wieder den schritt vom „was?“ zum „wie?“ zu vollziehen; dabei verwendet er die erkenntnisse der physik, chemie und anderer wissenschaften und hat im übrigen zunächst freie hand. Beispielsweise könnte die aufgabe für einen maschinenbauer lauten, ein automatik-getriebe für einen PKW zu konstruieren, das bei allen betriebszuständen den jeweils optimalen gang auswählt. Geht man einmal davon aus, dass durch weitergehende angaben die zu lösende aufgabe präzise genug beschrieben ist (dass also z. b. eindeutig festgelegt ist, in welcher beziehung der ausgewählte gang optimal sein soll), so bieten sich unübersehbar viele möglichkeiten an, wie man diese aufgabe theoretisch lösen könnte. Der ingenieur muss diese möglichkeiten überblicken, untersuchen und auf ihre praktische verwirklichbarkeit prüfen. Forschung in den zugrundeliegenden wissenschaften betreibt er dabei nicht; die Encyclopædia Britannica sagt:

[3] Der begriff *retirement* wird normalerweise für menschen verwendet und bedeutet das ausscheiden aus dem berufsleben (ruhestand).

> *„Die funktion des wissenschaftlers ist es, zu wissen, aber die des ingenieurs ist es, zu tun. Der wissenschaftler erweitert den vorrat an verifiziertem, systematisiertem wissen über die tatsächliche welt; der ingenieur macht dieses wissen tragfähig für praktische probleme."*

Wirtschaftliche abwägungen Es lassen sich eine menge dinge bewerkstelligen, wenn man bereit ist, beliebig viel geld dafür auszugeben. Normalerweise ist man jedoch nicht gerne bereit, unverhältnismässig viel für dinge auszugeben, die nicht unbedingt notwendig sind. So entscheidet vielfach der preis über leben und tod eines produkts. Aufgabe des ingenieurs ist es daher nicht bloss, ein produkt zu entwerfen, das die gewünschte funktion *irgendwie* erbringt, sondern er muss unter allen denkbaren und praktisch durchsetzbaren lösungsmöglichkeiten die kostengünstigste auswählen.

Sicherheit Von jeher gehören sicherheitsüberlegungen zu den pflichtübungen eines ingenieurs. Der bestimmungsgemässe gebrauch eines apparats darf weder den bediener noch den rest des systems gefährden, in das dieser apparat eventuell eingebaut ist. (Z. b. darf das automatikgetriebe nicht den motor beschädigen; es muss sich auch für den fahrer berechenbar verhalten, damit das fahrzeug sich nicht ungewollt in gang setzen kann usf.) Das erfordert eine ganz sorgfältige analyse der grenzfälle des betriebs sowie der grenzen der belastbarkeit des materials. Es gehört zum guten ton, dass bis zu einem gewissen grad auch der nicht bestimmungsgemässe gebrauch eines apparats oder bedienungsfehler entweder konstruktiv verhindert oder auch so sicher wie möglich gemacht werden. In jedem fall versucht der ingenieur durch gewisse sicherungen (überdruckventile, sollbruchstellen, elektr. schmelzsicherungen) einen denkbaren schaden möglichst gering zu halten. Sehr schön wird dies von Alexander Spoerl (anh. E, s. 239) beschrieben.

Typisch für die arbeit eines ingenieurs ist es also, dass er widersprüchliche anforderungen in einklang bringen soll. Wir zitieren aus dem auch für zukünftige software-ingenieure höchst lesenswerten buch (Dörner 1989, p. 97):

> *„In einer komplexen Situation ist es fast immer notwendig, sich nicht nur um ein Merkmal der Situation zu kümmern, also* ein *Ziel anzustreben, sondern man muß* viele *Ziele* gleichzeitig *verfolgen. Wenn man aber mit einem komplizierten, vernetzten System umgeht, so sind die Teilziele kaum je ganz unabhängig voneinander. Es ist vielmehr oft der Fall, dass Teilziele in einem kontradiktorischen Verhältnis zueinander stehen."*

Auch die Encyclopædia Britannica sieht diese Problematik:

> *„Anders als der wissenschaftler ist der ingenieur nicht frei in der auswahl der probleme, die ihn interessieren; er muss die probleme so lösen, wie sie sich stellen; seine lösung muss widersprüchliche anforderungen erfüllen. Normalerweise kostet effizienz geld; sicherheit erhöht die komplexität; verbesserte leistung erhöht das gewicht. Die lösung des ingenieurs ist*

die optimale lösung, das endresultat, das unter berücksichtigung vieler faktoren das wünschenswerteste ist. Es kann das zuverlässigste innerhalb einer gegebenen gewichtsgrenze sein, das einfachste, welches gewisse sicherheitsanforderungen erfüllt, oder das effizienteste zu gegebenen kosten."

Eine sehr lesenswerte diskussion von entwurfsentscheidungen findet sich auch bei Colwell (2004), der als chefentwickler der Pentium-prozessor-reihe mit vielen ingenieursfragen beschäftigt war. Colwell schreibt unter anderem:

> *„Eins der dinge, welche die tätigkeit des ingenieurs so interessant machen – abgesehen von der tatsache, dass man dafür bezahlt wird – ist das wechselspiel zwischen so vielen widersprüchlichen einflüssen: merkmale, leistung, technik, risiko, zeitplan, fähigkeiten des entwicklerteams und wirtschaftlichkeit. Du kannst eine perfekte abstimmung zwischen diesen allen finden und trotzdem erfolglos bleiben, weil du etwas entworfen hast, das die käuferschicht zufällig nicht haben will."*

1.3 Software ist anders

Versucht man all dieses fortzudenken auf entwurf und entwicklung von programmsystemen, so merkt man sehr bald, dass sich softwaretechnik grundsätzlich von den klassischen ingenieurdisziplinen wie etwa maschinenbau, elektrotechnik, bauwesen unterscheidet:

1. Die klassischen ingenieurdisziplinen haben naturwissenschaftliche erkenntnisse über die *naturgesetze* zur grundlage; diese naturgesetze kann der ingenieur nicht überwinden. Insofern gibt es eine kontrolle der ingenieurtätigkeit: Wer das unmögliche versucht, wird durch die naturgesetze eines besseren belehrt. Die notwendigkeit der erlernung und beherrschung der naturwissenschaftlichen und mathematischen grundlagen ist allgemein derart anerkannt, dass niemand sich ohne die erforderliche ausbildung als ingenieur betätigen wird; andererseits gehören mathematik, physik, chemie sowieso wegen ihrer allgemein anerkannten nützlichkeit und notwendigkeit zum unabdingbaren standardprogramm der sekundar-schulausbildung.

   Demgegenüber gibt es auf dem software-sektor scheinbar keine naturgesetze; alles, was denkmöglich ist, erscheint auch machbar. Wer hier das unmögliche versucht, kann mit etwas glück ziemlich lange damit durchkommen. Charakteristisches merkmal von software-systemen, die das unmögliche versuchen, ist ihre gewaltige und ständig zunehmende komplexität. Wir fühlen uns an einen ausspruch von Alan Perlis (1982) erinnert: *„In seeking the unattainable, simplicity only gets in the way"*.

   Dabei ist es keineswegs so, dass es für die software-erstellung keine grundlagen einer den naturgesetzen gleichkommenden qualität gäbe; die

theoretische informatik beweist, zurückgehend auf Gödel und Turing, dass bestimmte fragen aus prinzipiellen erwägungen heraus unmöglich durch ein programm beantwortet werden können; von anderen fragen, die theoretisch lösbar sind, beweist sie, dass sie einen prohibitiv grossen rechenaufwand bedingen. Solches grundlagenwissen ist aber von völlig anderer art als physikalische gesetze. Auch ist es unter den programmierern noch nicht überall verbreitet. Bei den klassischen ingenieurwissenschaften achten die standesvereinigungen und der gesetzgeber streng darauf, dass mit dem titel „Ingenieur“ ein ganz bestimmtes qualitätsniveau verbunden ist. Im softwarebereich gibt es dagegen zahlreich leute, die glauben, dass bereits die beherrschung der syntax einer programmiersprache sie zu programmierern macht. In vieler hinsicht ähnelt die heutige softwarepraxis dem zustand der ingenieurwissenschaften zur renaissancezeit, wo mit unglaublicher imagination und raffinesse immer wieder versucht wurde, ein „perpetuum mobile“ zu konstruieren und wo man das nicht-funktionieren der apparaturen auf die mangelnde handwerkliche qualität der ausführung zurückführte. Wenn man den zweiten hauptsatz der thermodynamik einmal verstanden hat, sieht man dagegen die fruchtlosigkeit dieser bemühungen sofort ein. Wenn heute auf dem softwaresektor ähnlich unmögliches versucht wird und natürlich nicht funktioniert, schiebt man die ursache auf ein paar fehlende megabyte hauptspeicher oder ein paar fehlende megahertz prozessortaktfrequenz.

2. Im softwarebereich haben wir es mit einer gänzlich *immaterialen technologie* zu tun. Software ist grundsätzlich verschleissfrei im gegensatz zu praktisch allen anderen ingenieursprodukten. Ein *produktionsprozess* bzw. eine *serienfertigung* im vergleich zum entwurf bzw. der anfertigung von prototypen ist hier nicht erkennbar. Entwurf und produktion fallen exakt zusammen, sieht man einmal von dem rein technischen vorgang der vervielfältigung von datenträgern und handbüchern ab. Das fehlen von abnutzung an der software könnte zunächst als vorteil gesehen werden, ist jedoch auch als nachteil zu begreifen, denn es impliziert, dass software in weitaus grösseren zeiträumen geplant werden muss, um unter wechselnden bedingungen immer noch einsatzfähig zu sein. Teilweise wird die abnutzung von software ersetzt durch die sich ändernden bedingungen im umfeld, in welchem die software eingesetzt wird. *Anpassbarkeit* ist hier ein ganz wesentliches kriterium. Ein dokumentiertes beispiel hierzu ist das sogenannte „Jahr-2000-problem“, das grosse aufregung erzeugt hat und grosse summen an geld verschlungen hat: Alte software, die zum teil noch aus den 50er und 60er jahren stammte, hatte jahreszahlen in der regel zweistellig codiert. Dass dies in der datumsarithmetik zu problemen beim jahrhundertwechsel führen kann, wurde damals nicht gesehen, da kein mensch damit gerechnet hatte, dass diese software auch nach einem

zeitraum von 40 jahren immer noch eingesetzt würde.

3. Den unterschied zwischen der *analogen welt* der klassischen ingenieure und der digitalen der softwareingenieure erwähnten wir schon. Diese bemerkungen würden im grunde auch auf die ebenfalls digitale computer-hardware zutreffen, die jedoch im gegensatz zur software als ausgesprochen zuverlässig gilt. Wie Parnas (1985) aber richtig bemerkt, ist die computer-hardware hochgradig repetitiv ausgelegt, d.h. sie besteht aus vielen kopien sehr kleiner digitaler subsysteme. Jedes dieser subsysteme kann analysiert und erschöpfend getestet werden, sodass ein erschöpfender test der gesamten hardware nicht nötig ist. Dieser redundanzeffekt der im raum abgewickelten repetitiven hardwarestruktur muss in der software u.u. anders bewerkstelligt werden. Die klassischen methoden der ingenieure, systeme so lange in teilsysteme zu zerlegen, bis man auf einem niveau angelangt ist, wo die korrektheit sozusagen ins auge springt, funktioniert bei software nicht: man sieht nicht, „wie die zahnräder ineinandergreifen“ und ganz entfernt liegende stücke der software können zu ungeahnten interaktionen führen.

4. Für die praktische arbeit des softwareingenieurs kommt ausserdem erschwerend hinzu, was sich in dem slogan *„software is soft“* ausdrücken lässt: die scheinbar leichte und kostenlose änderbarkeit von software. Wenn ein bauingenieur ein bestimmtes gebäude geplant und durchgerechnet hat, ist spätestens in dem moment, wo die fundamente oder die ersten betondecken gegossen sind, klar, dass man nicht noch zwei zusätzliche stockwerke aufsetzen oder eine tragende wand weglassen oder versetzen kann, ohne dass sehr hohe kosten für den abriss von bereits gebautem sowie grössere verzögerungen auftreten. In der softwarebranche ist es jedoch noch durchaus üblich, aufgaben, ziele und umfang eines systems oft selbst im letzten moment radikal zu ändern, weil es ja scheinbar so einfach ist: es brauchen ja bloss ein paar softwaremodule umgruppiert, geändert bzw. neu geschrieben werden; das ganze produkt steht ja letztlich immer nur „auf dem papier“ bzw. auf dateien.

1.3.1 Die arbeitswelt des softwareingenieurs

Sicher ist die frage berechtigt, inwiefern diese vorlesung etwas mit dem berufsleben des informatikers zu tun hat, denn die folgenden beobachtungen sind nicht von der hand zu weisen:

1. *Die anpassung existierender software kommt häufiger vor als das schreiben neuer systeme.* Software ist teuer und wird von den firmen, die sie einsetzen, als investitionsgut betrachtet. Es mag für den PC-hobbyisten befremdlich

wirken, dass beispielsweise im rahmen des „Jahr-2000-Problems“ herauskam, dass es grosse mengen an software im einsatz gab, die fast 50 jahre alt war (und in diesem zeitraum natürlich häufig den neuen anforderungen angepasst wurde). Deshalb haben softwareingenieure viel mehr mit der anpassung existierender programme zu tun als mit dem schreiben neuer programme. Daraus ergibt sich, dass sie nicht nur in der lage sein müssen, programme in der jeweils modernsten sprache zu *schreiben*, sondern sie müssen auch programme in allen möglichen (und teilweise abstrusen) programmiersprachen *lesen* und verstehen können. (In zweiter konsequenz ergibt sich daraus, dass ein echter profi programme stets so schreibt, dass sie auch von anderen wirklich gelesen und verstanden werden können; von cleveren tricks (*hacks*) ist abzuraten.)

2. *Beim schreiben neuer systeme entfällt der grösste marktanteil auf sog. standardsoftware, der grösste arbeitsaufwand auf individuell gefertigte systeme.* Wir beschäftigen uns in dieser vorlesung praktisch nur mit der *individuellen anfertigung von software* auf der basis spezifischer anforderungen eines auftraggebers, wohl wissend, dass das grosse geschäft nicht mit dieser art von software gemacht wird, sondern mit sog. *standardsoftware*, oft auch wegen ihrer verpackungsform *shrink-wrap software* genannt. Schaut man jedoch nicht auf das verdiente geld (den umsatz), sondern auf die *anzahl von projekten*, so sind die individuell auf kundenanforderung geschriebenen softwarepakete wieder in der überwältigenden mehrzahl.

   Dies trägt eine gewisse analogie zu der art und weise, wie normalerweise häuser gebaut werden oder auch zur massschneiderei von kleidung. Auf die analogie der softwaretechnik zur architektur werden wir noch zu sprechen kommen; an dieser stelle möchten wir zunächst die analogie zur textilindustrie betrachten. Sie ist besonders interessant, da heute praktisch gar keine kleider mehr massgeschneidert werden. Statt dessen sind wir es gewohnt, konfektionsware einzukaufen. Wenn man sich gedanken darüber macht, wie das system der konfektionskleidung funktioniert, so stellt man fest, dass diese tatsächlich eine bahnbrechende und ganz erstaunliche entwicklung darstellt. Hier fliesst eine über jahrhunderte gewachsene und über jahrzehnte erprobte kenntnis über gewisse proportionen des menschlichen körpers ein, sodass über 90% der menschheit mit konfektionskleidung ohne irgendwelche modifikationen auskommt. Der vorteil für die textilindustrie liegt dabei darin, dass eine stark arbeitsteilige produktion von kleidungsstücken mit maschineller unterstützung und teilweise weniger qualifiziertem personal ermöglicht wird, wobei die maschinen bzw. arbeitsabläufe nur relativ selten umgestellt werden müssen.

   In der softwarebranche gibt es kaum nennenswerte produktionskosten; ein ähnlicher trend ist trotzdem auch hier erkennbar. Die firmen, wel-

che das meiste geld auf dem softwaremarkt machen, leben nicht von der „massschneiderei“, sondern vielmehr von der konfektionsware. Auf der basis von marktforschungsdaten oder durch geniale intuition werden die bedürfnisse grosser kreise von computerbesitzern vorweggenommen bzw. inferiert und eine software hergestellt, die möglichst vielen anforderungen gerecht wird. Der grund für den erfolg der standardsoftware liegt, wie wir noch sehen werden, darin, dass die präzise ermittlung der anforderungen wesentlich schwieriger und teurer ist als der eigentliche programmierprozess. Deshalb kann standardsoftware viel billiger als individuell angefertigte software sein, weil für jede teuere anforderungsanalyse grosse stückzahlen identischer software verkauft werden können.

Durch die massive marktmacht der *shrink-wrap software* tritt die bedeutung der individuell nach bestimmten anforderungen hergestellten software scheinbar in den hintergrund. Trotzdem sind die in dieser vorlesung vermittelten kenntnisse aus zwei gründen wichtig:

1. Sehr vieles von dem, was hier zu besprechen ist, z. b. modularisierung, qualitätssicherung, schnittstellendefinitionen etc. bezieht sich auf individuell hergestellte software in gleichem masse wie auf standardsoftware.
2. Dort, wo unterschiede auftreten, werden diese teilweise dadurch kompensiert, dass durch das vordringen der standardsoftware die notwendigkeit nach experten grösser wird, welche in der lage sind, diese standardsoftware zu konfigurieren und anderweitig an bedürfnisse des benutzers anzupassen[4] (vergleiche dies mit den änderungsschneidereien). Hier werden dann jedoch genau die fähigkeiten und fertigkeiten relevant, die wir zuvor bei der produktion der *shrink wrap software* ausgeklammert haben.

1.4 Programmieren im grossen vs. programmieren im kleinen

Ein gängiges missverständnis ist die verwechslung von softwaretechnik mit programmierkunst. Sicherlich ist programmierkunst wichtig, das erlernen eines soliden fundaments an datenstrukturen und algorithmen ebenso wie das entwickeln einer gewissen virtuosität in der handhabung einer möglichst grossen anzahl von programmiersprachen. Trotzdem kann all dies die probleme der praxis nicht lösen, weil die schiere grösse von softwaresystemen ganz andere probleme schafft als diejenigen, die von programmierkünsten gelöst werden. Zwar gibt es auch einsame rufer in der wüste (Wirth 1995), welche die

[4]So berichtet das Handelsblatt Nr. 123 vom 1.7.1998 auf s. 55: „Viele neue Jobs entstehen bei Beratern und Integratoren. Im Wachstumsmarkt Information und Kommunikation nimmt das Geschäft mit Standardsoftware die führende Rolle ein.“

Software	*Jahr*	*Umfang*	*Aufwand*
Apollo/Skylab software	1970	10 MLOC	2000 mannjahre
Space Shuttle software	1977	40 MLOC	8000 mannjahre
Windows 95	1996	15 MLOC	???
Windows 2000	2000	30 MLOC	???
Windows XP	2001	35 MLOC	???
Windows Vista	2007	50 MLOC	???

Tabelle 1: Umfang von softwarepaketen

zunehmende komplexität von softwaresystemen beklagen und deren notwendigkeit in frage stellen, aber das ändert nichts daran, dass die meisten interessanten systeme tatsächlich sehr gross sind. Tab. 1 liefert dazu einige zahlen. In dieser tabelle wird die einheit

LOC = *„Lines of Code“*

verwendet, die auch mit den üblichen präfixen als kLOC, MLOC etc. auftaucht. Es soll nicht verschwiegen werden, dass manche autoren, z. B. Sneed und Jungmayr (2011), einen grossen unterschied machen zwischen *anzahl von zeilen* im programm und *anzahl von anweisungen*, da es ja anweisungen geben mag, die mehr als eine zeile beanspruchen. Das sind im grunde aber spitzfindigkeiten, die ausserdem zu streitigkeiten führen können, was denn nun „eine“ anweisung ist: Betrachte etwa das folgende codesegment in Modula-2:

```
WHILE p # NIL DO
  p := p^.next
END
```

Ist dies nun *eine* anweisung? Oder zwei? Oder gar drei? Egal, wie man sich hier entscheidet, es ist in jedem fall eine *syntaktische analyse* des programms notwendig, um anweisungen zählen zu können. Da ist es wesentlich einfacher (z. B. mit dem Unix-hilfprogramm `wc`), die zeilen zu zählen.

Manche leute möchten aber auch klarmachen, dass kommentarzeilen (ebenso wie eingestreute leerzeilen) bei ihnen nicht zählen und verwenden deshalb statt LOC die bezeichnung

NCSS = *„Non-Commentary Source Statements“*

Das mag durchaus einen signifikanten unterschied machen, aber auch dazu ist zumindest eine rudimentäre syntaxanalyse notwendig.

Man verwendet die einheit LOC/NCSS immer noch gerne als mass für den *programmieraufwand*, auch wenn es inzwischen modernere masse gibt.

Verwendet man LOC dagegen als steuerungsgrösse für die unternehmensführung, um die *produktivität* einzelner programmierer zu bewerten, so wird das mass unsinnig. Programmierer würden dann versuchen, „zeilenschinder" zu werden und möglichst viele LOC abzuliefern. Wartungsmassnahmen, welche code verbessern, indem sie zeilen eliminieren, würden dadurch in missgunst gebracht. Ausserdem werden andere wichtige ziele, z.b. wiederverwendung von code, damit in gefahr gebracht. Besonders die OO-welt lebt bzw. wirbt sehr stark mit der wiederverwendung (s. das zitat von B. Meyer in anh. E, s. 242).

Eine oft zitierte „volksweisheit" der softwaretechnik besagt, dass ein programmierer etwa 100 LOC pro woche schreibt. Das klingt zunächst wenig, aber es beinhaltet natürlich die zeiten für vorbesprechungen, planung, entwurf, testen, dokumentation etc. Rechnen wir die woche zu 40 stunden, so ergeben sich 2,5 LOC/h; berechnet man dann die programmiererstunde zu 90,– EUR , so kostet eine programmzeile 36,– EUR. Es gibt auch noch andere zahlen; Drake (1996) von der National Security Agency nennt 7–8 LOC pro tag (also höchstens 40 LOC/woche) und dementsprechend kosten von 70 USD pro zeile. Interessanterweise ist diese „programmierleistung" in LOC/woche unabhängig von der programmiersprache. Zwar muss man etwa im assembler viel mehr zeilen schreiben als in einer höheren programmiersprache, um eine vergleichbare aufgabe zu erledigen, und diese zeilen sind ungleich weniger komplex als die der höheren programmiersprache, aber dafür sind diese zeilen wegen der verwicklung mit der maschinenarchitektur und des niedrigen abstraktionsniveaus viel schwerer zu erstellen und zu testen.

Tab. 1 enthält ausserdem angaben für sog. „mannjahre" für den programmieraufwand. Diese zahlen sind so entstanden, dass die anzahl der beschäftigten in einem projekt multipliziert wurden mit der anzahl der jahre, die für das projekt gebraucht wurden. Die zahl „2000 mannjahre" für die Apollo/-Skylab-Software bedeutet logischerweise nicht, dass hier eine person 2000 jahre programmiert hat, aber auch nicht, dass 2000 programmierer dies innerhalb eines jahres geschafft haben. Auf diese problematik gehen wir im nächsten Abschnitt („Brookssches gesetz") genauer ein. Für die Windows-familie habe lassen sich keine aufwandszahlen angeben, weil keine publiziert worden sind und weil man angesichts des umfangreichen wiederverwendeten alten MS-DOS- und Windows-code auch schwer den aufwand abschätzen kann.

In tabelle 2 ist die programmierung im kleinen nochmals mit der programmierung im grossen verglichen; diese tabelle ist aber *cum grano salis* zu lesen; es trifft nicht immer alles gleichermassen zu. Als grenzlinie zwischen „kleinen" und „grossen" programmen werden unterschiedliche zahlen genannt; ich würde die grenze etwa bei 5 kLOC ziehen.

Prog. im kleinen	*Prog. im grossen*
Genaue spezifikationen liegen vor bzw. werden nicht benötigt	Spezifikationen müssen erst erarbeitet werden
Entwicklung in einzelarbeit	Entwicklung im team
Einschrittige lösung	Lösung in vielen schritten
lösung besteht aus einer komponente	Lösung in viele komponenten zerlegt
Entwickler und benutzer einheitlicher personenkreis	Entwickler und benutzer ungleiche vorbildung
Kurzlebige programme, unikate	Langlebige programme, programmfamilien

Tabelle 2: Programmierung im kleinen—programmierung im grossen

Daraus folgt für die softwaretechnik, dass sie sich über die reine programmierkunst hinaus mit den folgenden themen beschäftigen muss:

- *Requirements engineering*, d. h. eine sammlung von techniken, um zunächst einmal herauszufinden, was genau der auftraggeber von dem zu erstellenden system erwartet.
- Spezifikationsmethoden, -formalismen, -sprachen
- Schnittstellendefinition, gruppenorganisation, kommunikationsregeln
- Projektmanagement, ermittlung des projektfortschritts, pläne, „meilensteine“
- Dokumentationstechniken auf unterschiedlichen Ebenen (benutzerhandbücher, technische dokumentation)
- Versionskontrolle, release management
- Schulungsprogramme
- Software support, problem management, hotline

Ein phänomen, das sich im softwarebereich besonders unangenehm darstellt, ist das stetige wachstum von softwarepaketen. Tab. 3 (Neville-Neil 2008a) liefert dazu vergleichsdaten. Hier ist es sicherlich auch interessant, die LOC-daten der betriebssysteme mit denen in tab. 1 zu vergleichen, wobei aber zu bedenken ist, dass ein „kernel“ kein ganzes betriebssystem ist.

Programm	*Version*	*Jahr*	*Dateien*	*LOC*	*% geändert*
Emacs	21	2003	2.586	1.317.915	
	22	2008	2.598	1.771.282	34%
FreeBSD kernel	5.1	2003	4.758	2.140.517	
	7.0	2008	6.723	3.556.087	66%
Linux kernel	2.4.20-8	2003	12.417	5.223.290	
	2.6.25-3	2008	19.483	8.098.992	55%

Tabelle 3: Wachstum von softwarepaketen

1.5 Das Brookssche Gesetz

If it takes 10 men so many days to build a wall, how long would it take 300,000? — The wall would go up like a flash of lightning, and most of the men could not have got within a mile of it.
If a cat can kill a rat in a minute, how long would it be killing 60,000 rats? Ah, how long, indeed! My private opinion ist that the rats would kill the cat. (Lewis Carroll, 1880)

Das zitat von Carroll macht deutlich, was im grunde genommen sowieso klar sein sollte: Messgrössen wie „mannmonate“ implizieren heimlich, dass hier ein „dreisatzverhältnis“ gilt; im richtigen leben trifft diese mathematische idealisierung aber häufig nicht zu. Fred Brooks (1975) hat dies in seinem buch plastisch vorgeführt, s. auch sein zitat in anh. E, s. 239. Für den fall der softwareerstellung gilt nämlich, dass bei einer vollständigen trennung der teilaufgaben die entwicklungsdauer t umgekehrt proportional zur anzahl n der mitarbeiter wäre:

$$t \sim 1/n,$$

(s. abb. 3). Leider lassen sich aber die teilaufgaben fast nie völlig unabhängig darstellen, sodass mehr oder weniger kommunikation zwischen den mitarbeitern notwendig wird. Bei n mitarbeitern gilt für den kommunikationsaufwand k und eine konstante c im schlimmsten fall („jeder muss mit jedem kommunizieren“)

$$k \sim \binom{n}{2} \approx c\frac{n^2}{2},$$

Durch superposition der beiden kurven (s. abb. 3) sieht man, dass ab einem gewissen punkt zusätzliche mitarbeiter die entwicklungsdauer wieder vergrössern.

Kontrollfragen

Die nachfolgenden fragen sollten sich ohne nachlesen im skriptum beantworten lassen:

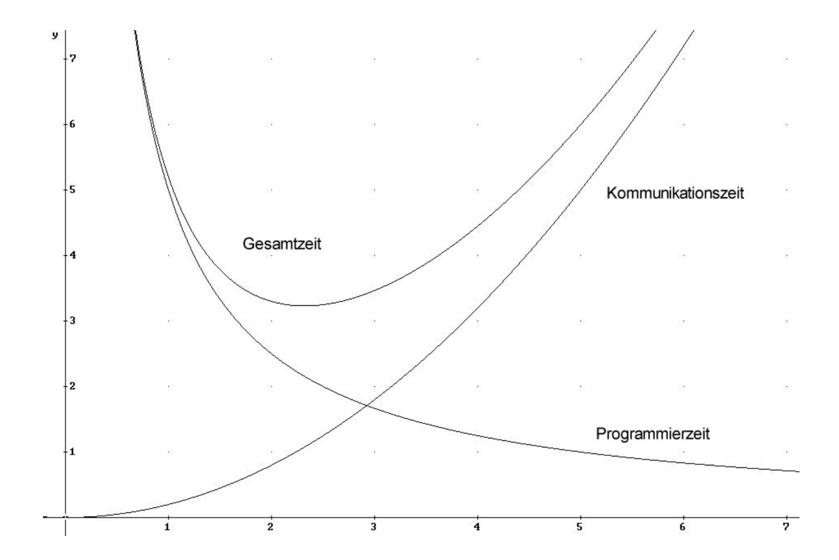


Abbildung 3: Mannmonate und minimale entwicklungszeit

1. Welches sind die drei hauptmerkmale der ingenieurstätigkeit?
2. Wodurch unterscheidet sich die herstellung von software von der herstellung anderer ingenieursprodukte?
3. Wodurch unterscheiden sich die charakteristika der programmierung im kleinen von denen der programmierung im grossen?
4. Wieviel LOC/woche schreibt ein programmierer durchschnittlich?
5. Wieso hängt die programmierleistung, wenn man sie in LOC/zeiteinheit misst, kaum von der verwendeten programmiersprache ab?
6. Inwiefern kann es unsinnig sein, die produktivität von programmierern in LOC/zeiteinheit zu messen?
7. Was genau besagt das Brookssche gesetz? Wie kommt es zu diesem sachverhalt?

2 Modulkonzept

Offensichtlich ist es unsinnig, grosse systeme mit vielen tausenden oder gar millionen von programmzeilen immer an einem stück zu übersetzen. Aus gründen der effizienz verfügten bereits die ersten ansätze zu höheren programmiersprachen über vorrichtungen zur getrennten übersetzung von programmstücken. Der compiler musste also für einen übersetzungsprozess nicht das gesamtprogramm kennen; die bestandteile wurden vom linker später zusammengesetzt unabhängig von der frage, aus welchen übersetzungsläufen sie entstammten. Damals hat man bereits getrennt übersetzte hauptprogramme bzw. unterprogramme als sog. *quellmodule*[5] bezeichnet, die daraus entstandenen dateien für die spätere verwendung durch den linker dementsprechend als *objektmodule*. Für unsere heutigen begriffe ist dies ein viel zu schwacher modulbegriff, da die verantwortung dafür, dass die einzelteile auch wirklich zusammenpassen, ausschliesslich beim programmierer ruht; das system kann hier keinerlei hilfestellung leisten.

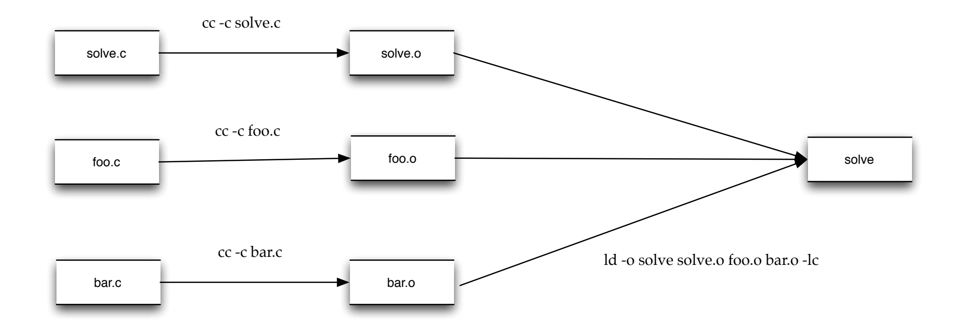


Abbildung 4: Getrennte übersetzung

Die situation ist in abb. 4 dargestellt: Ein programm zur geleichungslösung besteht aus drei verschiedenen C-quellprogrammen, die vom C-compiler jeweils getrennt übersetzt werden. Der linker `ld` setzt diese zu einem ausführbaren programm zusammen. Dabei geht auch die C-bibliothek `libc.a` ein, in der häufig benutzte unterprogramme in objektform bereitgestellt werden.

Von der rein technischen gegebenheit der getrennten übersetzung abgesehen, existiert ausserdem in jedem fall die notwendigkeit, systeme in bestandteile aufzuteilen, da sie als ganzes in der regel nicht verstanden werden können. In der art und weise, wie systeme in module strukturiert werden, liegt eine kreative und wichtige leistung; je nachdem, wie angemessen dies durchgeführt wurde, kann das verständnis des ganzen systems erleichtert, erschwert

[5] Es heisst übrigens *das* modul, pl. die module.

oder gar unmöglich gemacht werden.

Im vorgriff auf spätere teile der vorlesung soll erwähnt werden, dass softwareprojekte mit einer *analysephase* beginnen, in welcher die funktionalen anforderungen des zu erstellenden systems ermittelt werden, aber auch nichtfunktionale anforderungen wie die im anhang B aufgelisteten. Eine phase des systementwurfs (abschnitt 3) entwickelt eine *systemarchitektur*, die das in der problemspezifikation festgelegte system implementiert. Hauptproblem dabei ist die bewältigung der zugrundeliegenden komplexität. Es erscheint deshalb natürlich, das system in teilsysteme zu zerlegen, die leichter überschaubar und implementierbar sind.

Eine „vernünftige“ modularisierung wird etwa unter dem gesichtspunkt der anpassbarkeit alle die daten und funktionen, die von einer wahrscheinlichen änderung betroffen sein könnten, nach möglichkeit in einer abgeschlossenen systemkomponente in einer art und weise zusammenfassen, dass die restlichen strukturkomponenten davon unabhängig sind. Sie minimiert dadurch auch den kommunikationsaufwand zwischen den entwicklern verschiedener teile, indem sie funktionen und daten nach inhaltlichen kriterien zu modulen zusammenpackt.

Modularisierung dient also

- der bewältigung von problem- und programmkomplexität, da probleme und aufgaben in teilprobleme und -aufgaben zerlegt werden,
- der verbesserung der wartbarkeit, indem funktionen so lokalisiert werden, dass änderungen voraussichtlich in nur einem modul lokal durchgeführt werden können.
- der bewältigung des arbeitsumfanges, da die entwicklung verschiedener programmteile *parallelisiert* werden kann (von verschiedenen abteilungen/firmen/einrichtungen parallel durchgeführt werden kann),

In Meyer's Enzyklopädischem Lexikon findet sich die folgende definition des begriffs „modul“:

> *...ein austauschbares, komplexes Teil eines Geräts oder einer Maschine, das eine geschlossene Funktionseinheit bildet*

Im softwarebereich ergibt es natürlich kaum einen sinn, von dem gerät bzw. der maschine zu sprechen. Es bleiben aber die drei wesentlichen merkmale eines moduls:

- austauschbarkeit ohne beeinträchtigung des restsystems

- komplexe leistung
- geschlossene funktionseinheit

Der begriff „geschlossene funktionseinheit“ ist dabei so zu interpretieren, dass es möglich sein muss, die funktion des moduls innerhalb des gesamten systems sprachlich zu beschreiben, m. a. w. es muss möglich sein, die rolle zu beschreiben, die das modul im system übernimmt. Ein gutes indiz für diese tatsache ist es schon, wenn es gelingt, dem modul einen aussagekräftigen namen zu geben. Gelingt dies nicht, drängt sich der verdacht auf, dass hier ohne zwingenden grund irgendwelche überhaupt nicht zusammengehörenden funktionen und daten willkürlich als pseudomodul zusammengeklammert wurden. Mehr zu dieser thematik findet sich im abschnitt 3.

2.1 Geheimnisprinzip

Grundlegende gedanken zur modularisierung hat D. L. Parnas (1972) vorgestellt. In dieser oft zitierten arbeit wird das „geheimnisprinzip“ (*information hiding principle*) als von ihm bevorzugtes kriterium allerdings nur anhand eines beispiels (KWIC-index) erläutert und mit einem standardkriterium verglichen.

Die Parnas’schen ideen zum geheimnisprinzip lassen sich ungefähr folgendermassen zusammenfassen:

1. Ein modul kommuniziert mit seiner umwelt nur über eine klar definierte *schnittstelle*.
2. Das verwenden von modulen verlangt keine kenntnis ihrer inneren arbeitsweise.
3. Die korrektheit eines moduls kann man ohne kenntnis seiner einbettung in das gesamtsystem nachprüfen.

Dieses geheimnisprinzip, auch als „kapselung“ bekannt, ist der kern einer vernünftigen abstraktion und deshalb auch die hauptidee bei den *abstrakten datentypen* (abschnitt 3.3). Überhaupt ist es eins der wichtigsten konzepte der softwaretechnik. Inzwischen scheint es leider weitgehend in vergessenheit geraten zu sein, und zwar trotz aller publikationen im OO-sektor und trotz der tatsache, dass die kapselung die wichtigste eigenschaft des objektbegriffs ist[6]. Tatsache ist, dass systeme, die wirklich verwendet werden, einem stetigen wandel unterworfen sind[7]; die notwendigen änderungen lassen sich aber

[6]Objekte verfügen allerdings ausserdem über das konzept der *vererbung*, das in gewissem sinne dem geheimnisprinzip widerspricht, s. abschnitt 4.

[7]Schon 500 v. Chr. formulierte der griechische philosoph Heraklit die erkenntnis, dass das einzig beständige der wandel sei.

bei einer vernünftig eingeführten und rigoros eingehaltenen abstraktion viel leichter durchführen.

Bei einer aufteilung eines modularen systems auf mehrere programmierer bedeutet das geheimnisprinzip, dass jeder programmierer von den fremden modulen nur deren schnittstelle kennt, während die innereien vor ihm verborgen sind. Das gegenteil des geheimnisprinzips möchte ich Gorbačev zu ehren das *„glasnost-prinzip“* taufen. Es war seinerzeit von dem bereits zitierten F. Brooks für die entwicklung von OS/360 vorgeschrieben worden und findet heute noch anwendung bei Microsoft. Allerdings sagt Brooks (1995) nach 20 jahren: „Parnas was right and I was wrong“ und betont so die wichtigkeit des geheimnisprinzips.

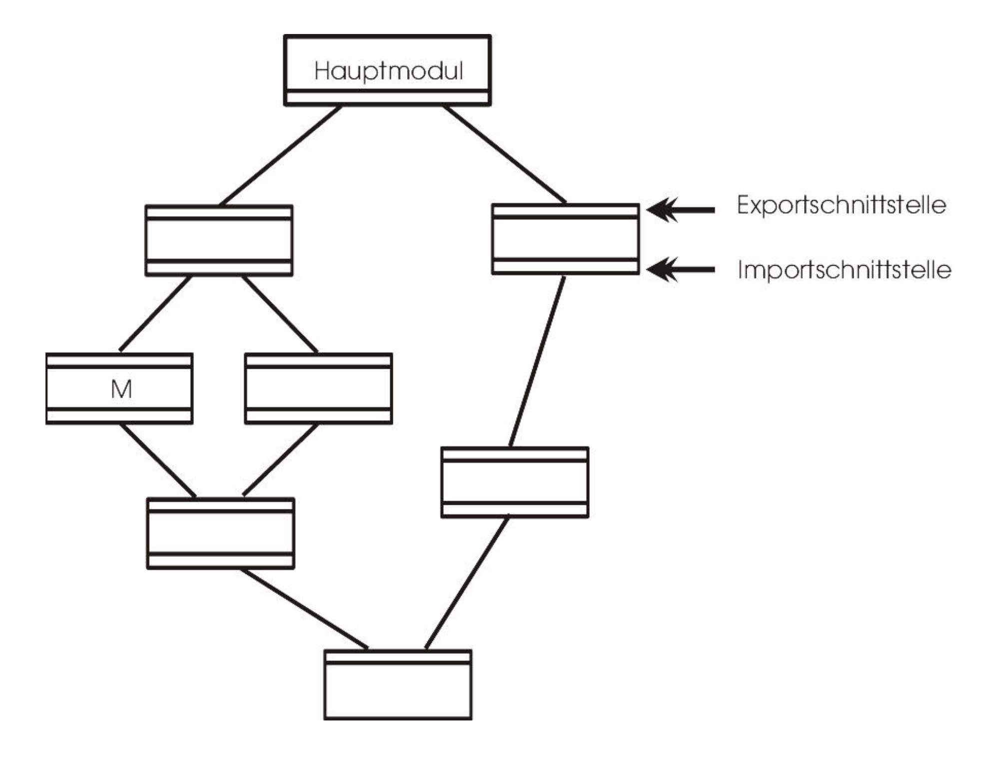


Abbildung 5: Ein modulares System

Eine typische situation im zusammenhang mit modularen systemen ist in abb. 5 dargestellt. In diesem bild haben wir die module durch kästen dargestellt, wie man es häufig tut, und dabei die exportschnittstelle durch einen schmalen streifen an der oberseite und die importschnittstelle an der unterseite markiert. Eine linie zwischen modulen charakterisiert dann eine import/exportbeziehung. Konzentrieren wir uns nun auf das modul M in der mitte der modulstruktur. Aus irgendwelchen gründen soll dieses modul neu geschrieben werden, sagen wir in form von M′. Wenn wir den modulgedanken ernstnehmen, müssen M und M′ kompatible schnittstellen haben (d. h. die export-

schnittstelle von M' muss die von M enthalten und die importschnittstelle von M' darf nicht über die von M hinausgehen) und in bezug auf die exportierten bestandteile von M die gleiche funktionalität aufweisen.

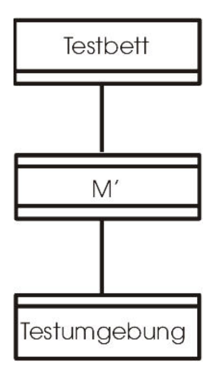


Abbildung 6: Testbett für ein Modul

Wollen wir also das neue modul M' auf einfache weise testen, ohne durch die komplexität des gesamtsystems belastet zu sein, so schreiben wir ein sogenanntes *testbett*, welches die funktionalität von M' in ausreichender weise prüft und hierüber ggf. protokoll führt. Diese situation ist in abb. 6 dargestellt. Damit wir diesen test überhaupt durchführen können, müssen wir auch die importe von M' korrekt bereitstellen, notfalls indem wir eine eigene *testumgebung* bereitstellen, welche diese funktionalität anbietet.

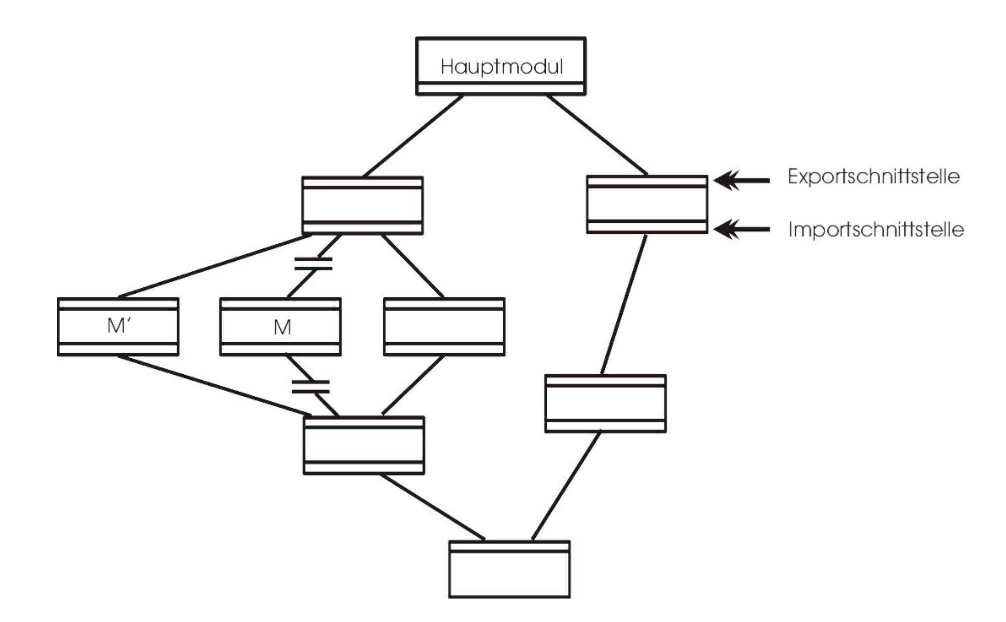


Abbildung 7: Austausch eines Moduls

Ist der test dann erfolgreich durchgeführt, so lässt sich im ursprünglichen system, wie in abb. 7 gezeigt, ohne weiteres das alte modul M gegen das neue modul M' austauschen.

2.2 Design by contract

Nimmt man das geheimnisprinzip ernst, so ergibt sich ziemlich zwangsläufig, dass die abmachungen im rahmen dieses geheimnisprinzips auch als elemente eines *kontrakts (vertrags)* zwischen einem anbieter (*supplier*) und einem kunden (*client*) angesehen werden können. Bertrand Meyer (1990, 1992) macht daraus eine generelle maxime für den softwareprozess, die er *design by contract* nennt. Ein kontrakt im üblichen sprachgebrauch hat zwei haupteigenschaften:

1. Jede partei erwartet *vorteile* aus dem kontrakt und ist bereit, *verpflichtungen* einzugehen, um diese zu erhalten.

2. Die vorteile und verpflichtungen sind in einem kontraktsdokument festgehalten.

Diesen gedanken überträgt Meyer auf die softwarekonstruktion, hier speziell auf die beschreibung von modulschnittstellen. Damit kein irrtum aufkommt: „kunde“ und „anbieter“ sind in diesem zusammenhang immer irgendwelche programmstücke (funktionen, methoden, prozeduren), keine menschlichen rollen.

Bertrand Meyer hat eine programmiersprache (Eiffel) entworfen, die es erlaubt, solche kontrakte zum bestandteil von programmen zu machen. Eiffel ist eine objektorientierte programmiersprache (s. kap. 4); deshalb heissen die funktionen bzw. prozeduren dort „methoden“. Technisch werden die kontrakte durch *logische formeln* beschrieben, von denen es drei unterschiedliche typen gibt:

Vorbedingung (*precondition*) Die vorbedingung einer methode drückt anforderungen aus, die jeder korrekte aufruf erfüllen muss. Wird eine methode mit werten aufgerufen, die nicht die vorbedingung erfüllen, so hat der kunde den kontrakt verletzt, die methode braucht nichts (sinnvolles) zu tun.

Nachbedingung (*postcondition*) Die nachbedingung drückt eigenschaften aus, welche im gegenzug durch eine ausführung der methode garantiert werden. Die garantie betrifft jedoch ausdrücklich nur den fall, wo die argumente die vorbedingung erfüllen.

Invarianten In der regel gibt es zwischen den einzelnen variablen eines moduls (einer „klasse“ bei Meyer) konsistenzbedingungen, und nach der reinen lehre des geheimnisprinzips ist dieser „zustand“ eines moduls nach aussen hin verborgen. Er kann nur vom modul selbst verändert werden, auskünfte über den zustand sind nur insofern möglich, als das modul selbst hierzu funktionen bereitstellt. Da die genannten konsistenzbedingungen über die zeit hinweg unverändert garantiert sein sollen, nennt man sie *invarianten* des moduls. Zu beachten ist, dass solche invarianten, solange das modul noch „am rechnen“ ist, d. h. also solange code aus diesem modul ausgeführt wird, selbstverständlich verletzt sein können – dies gilt auf einer anderen konzeptuellen ebene auch für schleifeninvarianten, die immer nur am schleifenkopf gültig sein müssen. Ist das modul aber inaktiv, so muss die invariante gelten.

Implizit gehört also die einhaltung der invarianten für jede einzelne methode zu den vor- und nachbedingungen: Die konjunktion von vorbedingungen und invarianten muss die konjunktion von nachbedingungen und invarianten implizieren.

Meyer wendet sich gegen die sogenannte „defensive programmierung“, bei der jeder kritische schritt durch geeignete abfragen abgeschottet ist, die dafür sorgen, dass nichts schlimmes geschehen wird. Er bemerkt dazu: „In vielen existierenden Programmen kann man die Inseln der nützlichen Verarbeitung kaum in den Ozeanen des Fehlerprüfcodes finden.“ Im zusammenhang mit dem „design by contract“ ist jedenfalls klar, dass der kunde dafür verantwortlich ist, die einhaltung der vorbedingung zu garantieren. Dies lässt sich unter anderem dadurch motivieren, dass an der aufrufstelle viel detailliertere kenntnisse über die funktionsparameter vorliegen und diese aufgabe deshalb dort viel leichter zu erledigen ist. Es gibt auch situationen, in denen eine überprüfung der vorbedingungen in der aufgerufenen methode den ganzen sinn der methode vernichtet: Beispielsweise ist eine vorbedingung für die *binäre suche*, dass das eingabe-array sortiert ist. Will man dies innerhalb der suche überprüfen, erübrigt sich das nachfolgende binäre suchverfahren. Welchen teil einer internen konsistenzbedingung man wirklich in die vorbedingung hineinschreibt und damit dem kunden als beweisverpflichtung aufgibt und welchen teil man dann doch innerhalb der funktion selbst überprüft, ist eine entwurfsentscheidung (s. anh. E, S. 241), aber der sogenannte *fordernde stil* (*demanding style*), bei dem die vorbedingungen möglichst aggressiv formuliert werden, hat sich in der praxis seiner ansicht nach bewährt.

Etwas spezifischer ist der fordernde stil durch die folgenden prinzipien charakterisiert:

Non-rendundancy principle Methoden prüfen niemals ihre vorbedingungen, da diese bereits vom aufrufer überprüft wurden.

Reasonable precondition principle Eine methode verlangt an vorbedingungen gerade so viel, wie sie für ihre arbeit braucht, aber nicht mehr. Die notwendigkeit der vorbedingung ergibt sich also aus der spezifikation.

Precondition availability principle Es muss möglich sein, die vorbedingung im namensraum des aufrufers zu formulieren, d. h. dem aufrufer sind alle begriffe verfügbar, die er für die überprüfung der vorbedingung braucht. Dies bedeutet insbesondere, dass keine details der implementierung verwendet werden können, da dies auch dem geheimnisprinzip widerspreche würde. (Es ergibt sich daraus natürlich logischerweise auch, dass dies für die nachbedingung ebenfalls gelten muss.)

Die frage des austauschens eines moduls M gegen ein modul M' können wir nun auch präziser diskutieren: M' muss den *vertrag* von M einhalten, das bedeutet im einzelnen:

1. Die vorbedingungen von M' sind schwächer als die von M: $\text{pre}(M) \Rightarrow \text{pre}(M')$
2. Die nachbedingungen von M' sind stärker als die von M: $\text{post}(M') \Rightarrow \text{post}(M)$
3. Die invarianten von M' machen die invarianten von M nicht ungültig (enthalten keinen widerspruch zu den invarianten von M)

Eine besondere beachtung verdienen in diesem zusammenhang auch die *ausnahmesituationen (exceptions)*: Eine methode, die feststellt, dass sie (aus irgendeinem grund) ihre nachbedingung nicht gewährleisten kann, kehrt nicht einfach an die aufrufstelle zurück, sondern wirft eine *exception*.

2.3 Bestandteile eines Moduls

Konzeptuell ist ein Modul durch die folgenden Komponenten bestimmt:

Importschnittstelle Hier wird festgelegt, welche konstrukte von ausserhalb hier verwendet werden. In der regel wird man dabei auch genau mitteilen, aus welchem anderen modul man diese konstrukte beziehen möchte; implizit wird natürlich auch der typ der konstrukte importiert.

Exportschnittstelle Hier wird festgelegt, welche konstrukte (funktionen, variablen etc.) aus diesem modul von ausserhalb benutzt werden dürfen; dabei wird gleichzeitig mindestens der *typ* dieser konstrukte mitgeteilt. Allgemein ist es wünschenswert, ausser dem typ noch weitere angaben über jede exportierte funktion zu spezifizieren:

Art der parameter Wird ein parameter nur gelesen oder auch geschrieben?

Seiteneffekte (eigentlich: nebenwirkungen, das wort „seiteneffekte“ ist eine unglückliche übersetzung von „side effects“) Welche globalen variablen werden von der funktion modifiziert?

Vorbedingungen, nachbedingungen entsprechend dem *design by contract*

Exceptions die ausnahmesignalisierungen, die von der methode geworfen werden können.

Invarianten Die invarianten des DBC können nicht zur exportschnittstelle gehören, da sie aussagen über den inneren zustand des moduls machen, der aber nach aussen nicht sichtbar ist.

Rumpf bestehend aus den exportierten funktionen, weiteren funktionen, die nur innerhalb des moduls zu verwenden sind und ggf. einem initialisierungscode, der zu beginn des programmlaufs die invarianten etabliert.

Oft wird vorgeschlagen, nur *funktionale schnittstellen* zu verwenden, d.h. nur funktionen zu exportieren. In der praxis ist es nämlich vielfach üblich, z.b. auch variablen zu exportieren; in einer funktionalen schnittstelle wäre dies nicht möglich, sondern es müsste ein paar von funktionen zum auslesen und verändern dieser variablen bereitgestellt werden. Der funktionale stil hat den vorteil, dass das modul selbst dafür sorgen kann, dass die invarianten erhalten bleiben. Ausserdem ist der funktionsaufruf ein relativ gut sichtbares ereignis, an dem protokollierungs- und testmassnahmen angehängt werden können, während die änderung eines variableninhalts praktisch vollkommen unbemerkt vonstatten gehen kann.

2.4 Spezifikation von software

Beim „design by contract“ spielt der begriff der *spezifikation* eine grosse rolle. Es ist nicht schwierig, einzusehen, dass in diesem falle spezifikationen umso wertvoller sind, je formaler (sprich: mathematischer) sie sind, denn nur die mathematische präzision liefert wirklich unmissverständliche spezifikationen. Später werden wir noch spezifikationen betrachten, die in früheren phasen der softwarekonstruktion anwendung finden, und diese werden kaum formal sein können. Prädikatenlogik zweiter stufe ist selten hilfreich im gespräch mit mathematisch nicht trainierten kunden. Deshalb sind die kontrakte, die zwischen softwareherstellern und ihren kunden geschlossen wurden, absolut unvergleichbar mit kontrakten im Meyer'schen sinne.

Im sinne des geheimnisprinzips ist es notwendig, dass spezifikationen einen gewissen freiraum für unterschiedliche implementierungen lassen; die von programmierern gelegentlich vertretene ansicht, dass der quelltext eines programms selbst die schönste spezifikation ist, widerspricht dem geheimnisprinzip diametral.

Was nun kommt, ist für informatiker keine überraschung: es gibt spezielle spezifikationssprachen, in denen sich der mathematische formalismus maschinenlesbar notieren lässt und die teilweise auch über dedizierte werkzeuge zur überprüfung ihrer (typ-) korrektheit verfügen. Bekannte beispiele sind VDM-SL (Jones 1990) und Z (Bowen 2005). VDM-SL ist die spezifikationssprache („specification language") des VDM-Projekts („Vienna Development Method"), das am Wiener labor von IBM begonnen wurde. Z stammt von der Oxford University und ist deutlich „mathematischer" angelegt.

Natürlich ist die frage berechtigt, wieso eine eigene spezifikationssprache notwendig ist, wenn doch im prinzip die ganze sprache der mathematik zur verfügung steht. Als antwort hierauf biete ich folgendes an:

1. Softwaretechnik verlangt die einhaltung von *normen* wie jede andere ingenieurwissenschaft. Die sprache der mathematik ist aber keineswegs normiert, sondern existiert in zahlreichen varianten.

2. Der immense reichtum der mathematischen formelsprache muss für spezifikationszwecke stark eingeschränkt werden, sonst werden nur vollausgebildete mathematiker in der lage sein, spezifikationen zu lesen.

3. Da spezifikationen maschinell behandelt werden sollen, mindestens aber maschinell speicherbar sein müssen, wird eine maschinenlesbare version der spezifikationen benötigt. Dazu ist es jedoch notwendig, den benötigten ausschnitt der mathematischen notation als sprache zu fixieren.

4. Unter umständen ist es sinnvoll, den formalismus einzuschränken auf konstrukte, die auch in der maschine eine rolle spielen (z. b. beschränkung auf endliche mengen, endliche abbildungen).

Ein relativ moderner ansatz unter verwendung der programmiersprache Java ist JML — Java Modeling Language (Leavens und Cheon 2006; Leavens 2007; du Bousquet u. a. 2004). JML ist inspiriert vom Larch-projekt (Guttag u. a. 1993) und ist in seinem anspruch noch etwas reduzierter als VDM-SL: die spezifikationen werden hier einfach innerhalb der programmiersprache Java formuliert, die nur durch wenige zusätzliche konstrukte erweitert wurde. Sie werden auch direkt innerhalb der zugehörigen Java-programme notiert – das sogenannte *single source*-prinzip – und dabei in eine spezielle form von kommentaren (`//@` für einen einzeiligen und `/*@...@*/` für einen mehrzeiligen

kommentar) eingepackt, damit sie den normalen übersetzungsprozess nicht stören. Diese eigenschaft teilt JML mit dem (noch zu besprechenden) Javadoc. Achtung: das `@`-zeichen darf von den vorausgehenden zeichen nicht durch eine leerstelle o. ä. abgetrennt werden! Die verwendung von JML wird dadurch nahegelegt, dass Java ab der version 1.4 sog. *assertions* unterstützt.

Dieses vorgehen hat folgende vorteile:

- Durch die beschränkung auf die möglichkeiten der programmiersprache braucht der programmierer nicht noch einen zusätzlichen formalismus zu erlernen.
- Das *single source*-prinzip sorgt dafür, dass die spezifikation immer an der stelle vorhanden ist, wo sie benötigt wird. Ausserdem kann – wie es die reine lehre vorsieht – die spezifikation ohne umstände *vor* dem programmierprozess erstellt werden; dazu werden einfach die spezifikationsteile zusammen mit den methodendeklarationen erstellt, wobei das eigentliche programm noch leer ist.
- Es sind werkzeuge verfügbar, welche die automatische behandlung solcher spezifikationen unterstützen, indem sie sich auf die implementierung der programmiersprache abstützen.

Im einzelnen kennt JML (unter anderem) die folgenden konstrukte:

- `requires` zur spezifikation von vorbedingungen,
- `ensures` zur spezifikation von nachbedingungen,
- `invariant` zur spezifikation von invarianten,
- `pure` zur spezifikation sog. *reiner* funktionen (methoden), d. h. funktionen ohne nebenwirkungen (*side effects*) und
- `modifies` bzw. `assignable` zur angabe von nebenwirkungen.

Die ersten drei konstrukte entsprechen genau denen von Eiffel, wo die jeweiligen bedingungen auch innerhalb der programmiersprache formuliert werden. Die bedingungen sind entweder

- kommentare der Form `(*...*)`, sogenannte *informelle beschreibungen* oder
- normale boolesche ausdrücke von Java. Dabei können diese ausdrücke u. a. durch die folgenden konstrukte ergänzt werden, die nicht zum eigentlichen Java gehören:

```
/*@ requires  n => 1 ;
  @ ensures \result ==
  @         (\product int i; 1<=i && i<=n; i);
  @*/
public static int factorial (int n) {
   int f = 1;
   int i = 1;
   /*@ loop_invariant i <= n && f == (\product int j;
     @               1 <= j && j <= i; j);
     @*/
   while (i < n) {
         i = i + 1;
         f = f * i;
   }
   return f;
}
```

Abbildung 8: Vor- und Nachbedingungen in JML

- `\result` bezeichnet den rückgabewert einer methode,
- `\old(E)` bezeichnet den wert des ausdrucks E vor dem methodenaufruf,
- `\exists` ist der existenzquantor. Die entsprechende syntax ist (`\exists` T x; P ; Q) mit der Bedeutung $(\exists x \in T) P \wedge Q$.
- `\forall` ist der allquantor. Die entsprechende syntax ist (`\forall` T x; P ; Q) mit der Bedeutung $(\forall x \in T) P \Rightarrow Q$.
- `\sum` bezeichnet die summe (`\sum` T x; P ; e) mit der Bedeutung $\sum_{x \in T \wedge P(x)} e$.
- `\product` bezeichnet das produkt (`\product` T x; P ; e) mit der Bedeutung $\prod_{x \in T \wedge P(x)} e$.

Abb. 8 zeigt ein beispiel einer JML-spezifikation für eine methode. Solche spezifikationen werden typischerweise einfach vor den Kopf der Methode geschrieben. Der kontrakt für die methode `factorial` ist, dass sie eine Zahl nimmt und ihre fakultät zurückgibt.

JML benutzt die klausel `requires`, um die obligation des clienten zu spezifizieren, und `ensures` für die obligation des implementierers. Die obligation des kunden ist es in diesem fall, eine positive zahl zu liefern; auf der anderen seite hat er das recht, die fakultät der zahl zu bekommen. In der gleichen weise darf der implementierer annehmen, dass das argument eine positive zahl ist, aber die obligation, die fakultät zurückzuliefern, muss er noch einhalten. Die vorbedingung einer methode muss zur aufrufzeit gelten. In diesem beispiel ist die

vorbedingung mit `//@ requires n =>1;` spezifiziert. Die nachbedingung der methode muss nach der terminierung der methode gelten. In diesem beispiel ist die nachbedingung:

```
 /*@ ensures \result ==
   @         (\product int i; 1<=i && i<=n; i);
   @*/
```

In JML können auch schleifeninvarianten mit `loop_invariant` spezifiziert werden. Die folgenden zeilen spezifizieren die schleifeninvariante der *factorial*-methode:

```
/*@ loop_invariant i <= n && f == (\product int j;
  @                1 <= j && j <= i; j);
  @*/
```

Es gibt noch weitere spezifikationsmöglichkeiten, u. a. für fehlerfälle („*exceptions*“), die aber in dieser vorlesung nicht angesprochen werden. Die spezifikation komplexer bedingungen wird erleichtert durch einige packages, die unter `org.jmlspecs.models` verfügbar sind.

Der „kontrakt“ im sinne des DBC besteht hierbei aus dem methodenkopf und den JML-spezifikationen, die ihm direkt vorangehen.

Als zusätzliches beispiel diene hier die spezifikation einer methode für die binäre suche in einem array. Voraussetzung für einen sinnvollen einsatz dieser methode ist, dass das array nicht leer ist und (o. b. d. a.) aufsteigend sortiert ist:

```
/*@ requires a != null
  @          && (\forall int i;
  @                      0 < i && i < a.length;
  @                      a[i-1] <= a[i]);
  @*/
int binarySearch(int[] a, int x) {
   // ...
}
```

In diesem beispiel ist auch sichtbar, wie die syntax des `\forall` gedacht ist: die variable, über die hier quantifiziert wird, muss selbstverständlich wie alles andere in Java typisiert sein. Die zweite komponente gibt eine einschränkung für den bereich, über den hier quantifiziert wird und die dritte enthält die eigentliche behauptung. Damit ist aber auch klar, auf welche weise hier code erzeugt werden kann, der die einhaltung der vorbedingung zur laufzeit überprüft: es wird einfach eine variable des angegebenen typs deklariert und dann läuft eine schleife über den angegebenen bereich, innerhalb deren rumpf die gültigkeit der behauptung überprüft wird. Die einschränkung kann

auch weggelassen werden, allerdings mit der konsequenz, dass dann kein automatischer prüfcode erzeugt werden kann. (Beachte, dass die ableitung einer schleife aus der hier angegebenen bedingung eine nicht-triviale leistung ist!) Eine ähnliche wirkung hat die angabe eines einschränkenden bereichs beim existenzquantor: hier wird auf wunsch code erzeugt, der alle elemente des bereichs darauf überprüft, ob sie die angegebene bedingung erfüllen. Wird keine gefunden, ist der existenzquantor fehlgeschlagen.

Ein beispiel für das zusammenspiel von vor- und nachbedingungen findet sich in der folgenden methode, welche das abheben eines (positiven) betrags von einem konto implementiert und dabei den neuen kontostand als ergebnis liefert:

```
/*@ requires amount >= 0;
    ensures balance == \old(balance)-amount &&
                       \result == balance;
  @*/
public int debit(int amount){
   ...
}
```

Wir finden in diesem beispiel übrigens eine verletzung des *precondition availability principle*: Der aktuelle kontostand `balance` wird in jedem fall eine private variable der kontoklasse sein und sich nur über entsprechende methoden verändern lassen. Deswegen kann er in der nachbedingung auch nicht sinnvoll verwendet werden.

In abb. 9 findet sich ein beispiel für eine „schwergewichtige“ (*heavyweight*) JML-spezifikation (Leavens 2007). Im gegensatz zu einer „leichtgewichtigen“ (*lightweight*) spezifikation, bei der nur *einige* aspekte spezifiziert werden, signalisiert das schlüsselwort `normal_behavior`, dass der programmierer davon ausgeht, dass er hier wirklich alle aspekte spezifiziert hat. Insbesondere darf die methode `largest`, wenn ihre vorbedingungen erfüllt sind, nicht mit einer ausnahmebedingung (*exception*) aussteigen, sondern muss unbedingt normal beendet werden. Die abbildung soll im folgenden noch detaillierter kommentiert werden:

Es geht darum, einen *heap* von ganzen zahlen zu spezifizieren. Zeile 5 sagt dazu, dass man sich diesen vorstellen kann als ein nicht-leeres feld ganzer zahlen. Die angabe `model` sagt dabei, dass dies hier keinen bezug zur implementierung haben muss: Diese darstellung wird nur benutzt, um die spezifikation herzustellen. Da das feld `elements` auch nur im JML-teil deklariert wird und nicht innerhalb des Java-codes, kann sich auch kein verwender des ADT `IntHeap` darauf beziehen. Die schwergewichtige spezifikation der methode `largest` sagt, dass diese methode nur aufgerufen werden darf, wenn mindestens ein element im *heap* gespeichert ist, dass die methode keine varia-

```
package org.jmlspecs.samples.jmlrefman;

public abstract class IntHeap {

    //@ public model non_null int [] elements;

    /*@ public normal_behavior
      @   requires elements.length >= 1;
      @   assignable \nothing;
      @   ensures \result
      @        == (\max int j;
      @                0 <= j && j < elements.length;
      @                elements[j]);
      @*/
    public abstract /*@ pure @*/ int largest();

    //@ ensures \result == elements.length;
    public abstract /*@ pure @*/ int size();
};
```

Abbildung 9: Heavyweight JML specification

blen modifiziert (d. h. also nebenwirkungsfrei ist) und dass sie in jedem fall das grösste element aus dem *heap* liefert. Die methode `size` ist im gegensatz zu `largest` *leichtgewichtig* spezifiziert; hier gibt es keine angaben zu ausnahmezuständen und vorbedingungen, nur die (fehlende) wirkung auf globale variablen ist durch das schlüsselwort `pure` angegeben. (Bei `largest` wurde dies redundanterweise auch noch spezifiziert, obwohl sich diese eigenschaft schon aus dem `assignable \nothing` ergäbe.

Für die behandlung von Java-programmen mit JML-spezifikationen stehen die folgenden werkzeuge bereit:

- Der JML-compiler `jmlc` dient als alternative zum normalen Java-compiler `javac`. Er liest und verarbeitet die JML-annotierten Java-programme und erzeugt daraus einen bytecode, der zusätzlich zu den normalen leistungen auch prüfcode für alle vor- und nachbedingungen und invarianten enthält.

- Das wrapper-skript `jmlrac` (*runtime assertion checker*) setzt die pfade für die JVM (`java`) so, dass in `jmlruntime.jar` auch die für JML notwendigen klassen gefunden werden, so dass die prüfung der bedingungen funktionieren kann.

- Das werkzeug `jmlunit` erzeugt aus den von `jmlc` verarbeiteten JML-annotierten Java-programmen automatisch testfälle zur verwendung mit dem populären Java-testwerkzeug *JUnit*.

- Der dokumentationsgenerator `jmldoc` erzeugt HTML-seiten, welche die spezifikationsanteile herauskristallisieren und lesbar aufbereiten.
- Der erweiterte syntaxprüfer `escjava2` (*extended static checker*) prüft möglichst viel von den bedingungen bereits zur übersetzungszeit nach, ohne dass dazu ein lauffähiges programm erzeugt werden muss.

ESC/Java2 führt die üblichen syntaxprüfungen des Java-compilers durch, inklusive verbesserter typprüfungen, verwendet aber ausserdem einen theorembeweiser, um mögliche denkfehler aufzudecken. Unter anderem werden stellen aufgedeckt, an denen möglicherweise eine null-referenz stattfindet; auch überschreitungen von array-grenzen werden, soweit möglich, bereits zur übersetzungszeit entdeckt:

```
class A{
  byte[] b;
  public void n() { b = new byte[20];}
  public void m() { n();
                    b[0] = 2;
                    ...  }
```

Hier würde ESC/Java2 sich bei der zuweisung auf `b[0]` mit einer warnmeldung dahingehend äussern, dass hier vielleicht zur laufzeit ein fehler auftreten wird, wenn `b` nicht richtig initialisiert wurde. Die tatsache, dass durch den aufruf von `n()` genau diese initialisierung vorgenommen wurde, ist für das werkzeug nicht einsichtig; dazu müssten ja auch die eigenschaften des konstruktors für `byte[]` inspiziert werden. Es ist deshalb richtig und nötig, die wirkung von `n()` ausdrücklich selbst zu beschreiben, z. b. durch ein

```
  //@ ensures b != null && b.length = 20;
```

vor der deklaration von `n()`. Für den angegebenen zweck hätte die erste hälfte dieser zusicherung genügt, aber wenn man schon einmal dabei ist, kann man auch gleich alles spezifizieren, was man weiss. Gebraucht wird es bestimmt in der einen oder anderen situation. Alternativ könnte man auch vor die genannte zuweisung ein

```
  //@ assumes b != null && b.length > 0;
```

einfügen.

2.5 Softwaretechniksprachen

In den sogenannten *softwaretechniksprachen* (*software engineering languages*), zu denen wir ausser Modula-2 auch Ada, Eiffel und Oberon rechnen können, gibt

es eine ausgefeilte kontrolle über die sichtbarkeit von objekten. Von einer softwaretechniksprache erwarten wir, dass sie über ein eingebautes modulsystem verfügt, das mindestens die folgenden eigenschaften hat:

1. Unterstützung von Software-abstraktionen durch mechanismen zur
   - Spezifikation einer schnittstelle,
   - Prüfung einer implementation gegen die spezifizierte schnittstelle, insbesondere auch auf vollständigkeit,
   - Prüfung des gebrauchs einer schnittstelle gemäss ihrer spezifikation,
   - Prüfung des konsistenten gebrauchs der gleichen version einer schnittstelle beim linken des programms
2. Kapselung von namensräumen, durch:
   - Konstrukte zur vermeidung von namenskonflikten (qualifizierte namen)
   - Gruppierung in namensräume (module).

Beachte, dass sich aus diesen eigenschaften unter anderem zwingend ergibt, dass der sprachprozessor (compiler) einer softwaretechniksprache syntaxprüfungen über die grenzen von übersetzungseinheiten hinweg vornehmen muss.

Besonders deutlich wird dies am beispiel der sprache Modula-2, die gerade mit dem ziel entworfen wurde, systeme als kollektionen von modulen sicher schreiben zu können. Die vorstellung bei Modula-2 ist, dass die schnittstellenbeschreibung eines moduls stets in einer anderen datei enthalten ist als die details der implementierung. Der quelltext eines moduls ist somit immer auf zwei dateien verteilt. Die datei mit der schnittstellenbeschreibung beginnt mit den schlüsselwörtern **DEFINITION MODULE**, die mit der implementierung beginnt mit **IMPLEMENTATION MODULE**. Modula-2-programmierer reden deshalb irreführenderweise vom *definitionsmodul* und dem *implementierungsmodul*, als ob dies zwei getrennte module seien. Wir verwenden hier auch diese terminologie, da sie sich leider etabliert hat. In einem definitionsmodul sind alle die objekte aus dem modul aufgeführt, die von anderen modulen verwendet werden können (exportschnittstelle); importe aus anderen modulen finden an dieser stelle nur insoweit statt, als sie dazu gebraucht werden, um die exportschnittstelle aufzuschreiben. Am beispiel einer einfachen druckerwarteschlange soll dies vorgeführt werden. In abb. 10 finden sich die schnittstellenbeschreibungen eines moduls *Jobs*, welches die eigenschaften eines druckauftrags einkapselt und einer druckerwarteschlange *PrintQueue*.

```
DEFINITION MODULE Jobs;
        (* Define the essential properties of a print job *)

CONST
     maxDescLen = 40;

TYPE
     job;
     desc  = ARRAY [0..maxDescLen-1] OF CHAR;

PROCEDURE priorityOf    (j: job): CARDINAL;
PROCEDURE descriptionOf (j: job; VAR d: desc);
PROCEDURE create        (d: desc; prio: CARDINAL): job;

END Jobs.
```

```
DEFINITION MODULE PrintQueue;
        (* A simple print queue *)

FROM Jobs IMPORT job;

TYPE   queue;

PROCEDURE new(VAR q:queue);
        (* Create a new print queue *)

(* Calls to the next three procedures are only valid after
   q has been created via new(q) *)

PROCEDURE enqueue(VAR q:queue; j:job);
        (* Enter a job j into queue q *)

PROCEDURE next(VAR q:queue):job;
        (* Get the next job from the queue, at the same
           time deleting it from the queue *)

PROCEDURE isEmpty(q:queue):BOOLEAN;
        (* Test for empty print queue *)

END PrintQueue.
```

Abbildung 10: Eine einfache druckerwarteschlange

Bemerkt werden muss hier, dass wir dieses beispiel textlich ganz knapp gehalten haben; zu jedem konstrukt wären hier normalerweise selbstverständlich eingehende kommentare fällig, welche die leistungen und einschränkungen davon beschreiben. Von *design by contract* ist bei Modula-2 keine rede; bestenfalls liessen sich diesbezügliche spezifikationen in kommentaren unterbringen. Das modul *PrintQueue* offenbart ausserdem noch eine andere eigenart, die sich häufig bei modulen findet: Die angebotenen funktionen sind nicht alle unabhängig voneinander, sondern setzen teilweise voraus, dass vor dem aufruf einiger funktionen zuerst andere aufgerufen werden. Im extremfall gibt es regelrechte *path expressions*, welche die korrekten aufrufreihenfolgen spezifizieren. In Modula-2 kann auch dies nur in kommentaren angedeutet werden.

Eine besonderheit hierbei sind die *opaken typen stack* und *queue*. Über deren darstellung wird gar nichts bekannt gemacht ausser dass es typen sind. Diese information wird auch unbedingt gebraucht, denn sonst lassen sich die schnittstellen der zugehörigen prozeduren gar nicht aufschreiben. Der verwender, der diese module importiert, kann deshalb mit variablen dieser opaken typen auch gar nichts „tun“ ausser sie den zugehörigen funktionen als argumente anzubieten. Das modul *Jobs* zeigt, wie funktionen[8] angeboten werden, die über die bestandteile einer verborgenen datenstruktur auskunft geben.

Definitionsmodule müssen wie implementierungsmodule auch vom Modula-2-compiler übersetzt werden; das ergebnis nennen wir eine *symboldatei*. Bei diesem übersetzungsprozess wird ein jeweils eindeutiger sogenannter *modulschlüssel* erzeugt, welcher die symboldatei mit dieser version der exportschnittstelle verbindet. Wie wir noch sehen werden, lässt sich auf diese art und weise später garantieren, dass bei der verwendung eines moduls dessen importierte schnittstelle auch wirklich diejenige der aktuellen implementierung ist.

Nachdem diese definitionsmodule compiliert wurden, lassen sich schon module schreiben, die diese verwenden, auch wenn noch keine einzige zeile des implementierungscodes geschrieben wurde. Der compiler kann aber bereits jetzt prüfen, ob die syntaktischen eigenschaften (vor allem die typen!) der schnittstellenspezifikation eingehalten wurden und kann auch schon objektcode erzeugen. Die implementierungen für die importierten module müssten sowieso später vom linker eingebunden werden; für das augenblicklich erzeugte objektmodul werden sie nicht gebraucht. Der Modula-2-compiler trägt in das erzeugte objektmodul für den linker ein, welche entitäten er aus welchen modulen benötigt *und* welches die modulschlüssel der fremden module gewesen sind, die er in den symboldateien vorgefunden hat.

[8] In Modula-2 heissen die funktionen auch **PROCEDURE**.

Wird nun ein implementierungsmodul compiliert (wozu zuerst das zugehörige definitionsmodul compiliert worden sein muss), so überträgt der compiler den modulschlüssel aus dem zugehörigen definitionsmodul in die objektdatei. Somit enthält jede objektdatei eine eindeutige charakterisierung der version von schnittstelle, die dazugehört.

Der Modula-2-linker trägt nun aus allen ihm übergebenen objektmodulen die modulnamen und die modulschlüssel zusammen. Wenn nun ein modul A ein modul B importiert und wenn zu dem zeitpunkt, wo das objektmodul zu A erstellt wurde, k_B der schlüssel von B war, und wenn dann später im zuge einer änderung der schnittstelle von B das objektmodul zu B einen anderen schlüssel k'_B erhält, so stellt der linker fest, dass diese beiden schlüssel nicht zueinander passen und meldet einen *versionskonflikt*. Das ist auch richtig so, denn das objektmodul zu A, welches von einer inzwischen veralteten schnittstelle von B ausging, ist mit grosser wahrscheinlichkeit durch die änderung der schnittstelle ungültig geworden. Der versionskonflikt kann nur behoben werden, indem A erneut compiliert wird, wobei möglicherweise durch die veränderung der schnittstelle von B nunmehr fehler vom compiler angemerkt werden.

Ein beispiel soll dies verdeutlichen: Angenommen, in dem `PrintQueue`-beispiel wird, nachdem das system fertig programmiert ist, nachträglich der deskriptor der jobs von 40 auf 20 zeichen verkürzt. Würde man nun einfach das system erneut zusammenlinken, gäbe es den klassischen *buffer overflow*: Ein deskriptor der länge 20 würde vom modul `PrintQueue` mit 40 zeichen überfüllt. Im falle von Modula-2 kann dies nicht passieren, denn der linker würde – ohne irgendetwas von der grösse von datenstrukturen wissen zu müssen – in jedem fall einen versionskonflikt anmerken, der sich nur dadurch auflösen liesse, dass das modul `PrintQueue` erneut übersetzt wird. Dann sind aber die deskriptoren auch auf 20 zeichen begrenzt, der *buffer overflow* kann sich nicht ereignen.

Auf den ersten blick erinnern die definitionsmodule an die aus der programmiersprache C bekannten *header files* (bzw. umgekehrt); in wirklichkeit wird von dem modulkonzept in Modula-2 jedoch wesentlich mehr geleistet: Während ein *header file* in C nur deklarationen von namen enthält, die dem compiler helfen können, ein vorliegendes programmstück zu übersetzen, und dabei völlig offen lassen, auf welche weise später linker und loader diese objekte auffinden sollen, wird in Modula-2 eine eindeutige zuordnung von namen von objekten (konstanten, typen, prozeduren) zu modulen vorgenommen. Im oben aufgeführten beispiel ist vollkommen klar, dass das modul *PrintQueue* den typ *job* aus dem modul *Jobs* importieren möchte und nicht irgendeinen anderen artikel gleichen namens. Damit ist vor allem unter anderem klar, in welchem objektmodul der linker später das symbol finden wird. Das C-

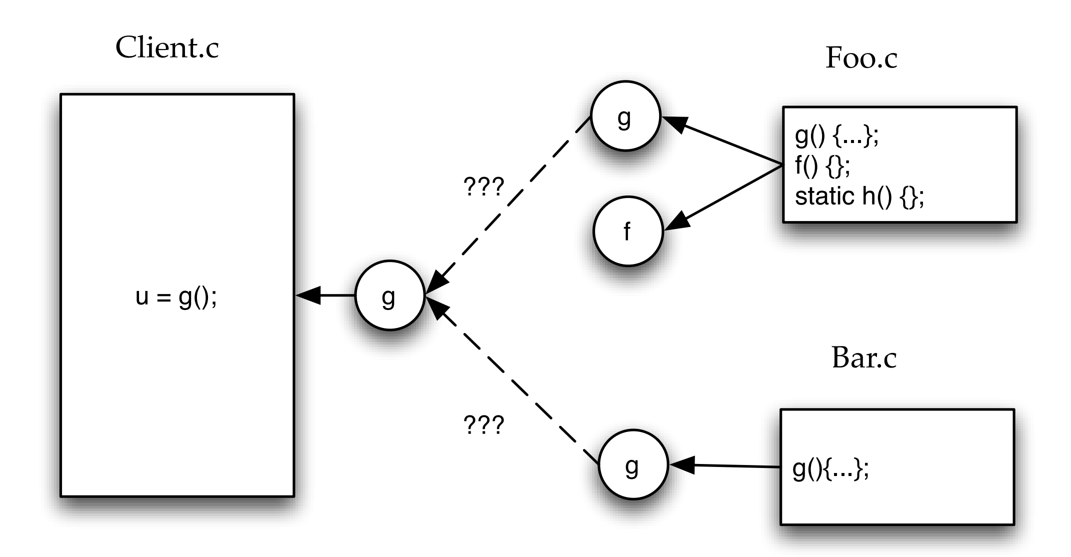


Abbildung 11: Namensräume in C

programmiersystem verfügt dagegen über einen *flachen* bzw. *globalen* namensraum: Alle symbole in einem objektmodul, die nicht ausdrücklich mit dem schlüsselwort `static` markiert sind, werden unverändert in einen *globalen namensraum* für den linker exportiert. Diese situation ist in abb. 11 veranschaulicht. Die übersetzungseinheit `Client` verwendet das symbol `g`, ohne es deklariert zu haben. Das stört den C-compiler bei der übersetzung von `Client.c` auch gar nicht; er reicht die information über das fehlende symbol einfach an den linker weiter. Wenn dieser später in keinem der mit einem aufruf zusammen übergebenen objektmodule und in keiner der spezifizierten bibliotheken das symbol finden kann, gibt es eine fehlermeldung. Da der C-compiler den typ von `g` nicht notwendigerweise kennt (die sogenannten „prototypen“ in den *header files* können unvollständig spezifiziert sein!) kann er auch nicht prüfen, ob `g` hier seinem typ entsprechend richtig verwendet wurde. Der linker kann umgekehrt, wenn er später ein verwendetes symbol nicht finden kann, nicht feststellen, woher es hätte kommen sollen. Findet er allerdings ein symbol doppelt, wie in abb. 11, so gibt er wiederum eine fehlermeldung aus. Kein problem entsteht dagegen wiederum, wenn ein symbol, das bereits in einem explizit übergebenen objektmodul enthalten war, in einer bibliothek nochmals exportiert wird; in diesem fall hat die explizite definition einfach vorrang vor der bibliothek.

Ein weiteres problem liegt in einem inkonsistenten gebrauch von headern. Nehmen wir an, `g` ist in `Foo.h` und `Bar.h` verschieden deklariert. Modul `Client` verwendet `Foo.h` und ruft `g()` auf, beim linken wird aber versehentlich `Bar.o`

angegeben, wobei `Bar.h` eine völlig andere definition für `g` enthält. In dem fall gibt es beim linken von `Client` mit `Bar` überhaupt keine fehlermeldung, aber das programm wird mit grosser sicherheit fehlerhaft sein.

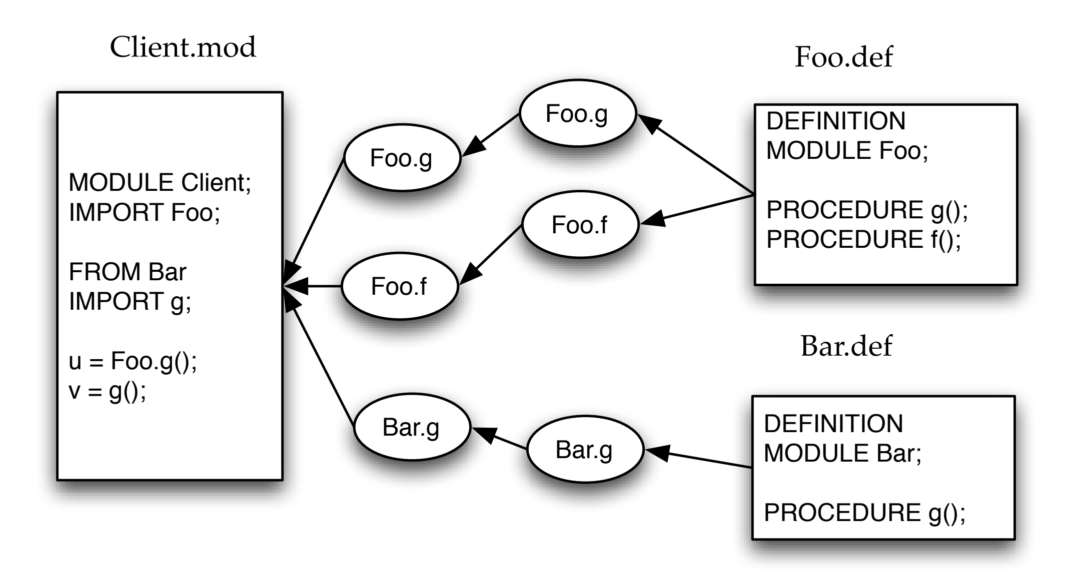


Abbildung 12: Namensräume in Modula-2

Ganz anders dagegen ist die situation in Modula-2. In abb. 10 haben wir ganz klar gemacht, dass der typ *job* aus dem modul *Jobs* kommen soll. Alle exportierten objekte eines moduls haben eine präzise typinformation, die vom compiler über die grenzen einer übersetzungseinheit hinweg wahrgenommen und geprüft wird. Exportiert ein modul in seiner schnittstellenbeschreibung ein objekt, das in der implementierung nicht vorkommt („vergessen wurde“), so gibt es beim übersetzen der implementierung eine fehlermeldung. Modula-2 hat überdies einen *strukturierten namensraum*: Jedes objekt wird beim export mit dem namen des moduls *qualifiziert*, aus dem es stammt. Diese situation ist in abb. 12 dargestellt.

In Modula-2 existieren zwei methoden, ein konstrukt in einem anderen modul verfügbar zu machen. In abb. 12 wurde für das symbol *g* der sogenannte *qualifizierende import* gewählt, d. h. ein vorkommen des bezeichners *g* im modultext wird automatisch mit dem bezeichner *Bar.g* identifiziert.

Dieses verfahren kommt der situation mit den *header files* von C am nächsten. Die hinter **FROM ... IMPORT** aufgeführten namen werden unmittelbar in den namensraum des vorliegenden moduls eingeführt, was eine bequeme kurze notation ermöglicht, aber andererseits natürlich auch zu namenskonflikten führen kann. (In C hätten wir durch `#include Bar.h` *alle* dort aufgeführten

namen bekannt gemacht, während wir bei Modula-2 noch eine teilauswahl treffen können.) Mit dem qualifizierenden import ist andererseits klar, dass das unter diesem namen importierte objekt nicht noch einmal von einer anderen stelle importiert oder gar im laufenden modul neu deklariert werden kann.

Eine andere möglichkeit, die vom standpunkt der wartbarkeit deutlich zu bevorzugen ist und die deshalb in der nachfolgersprache von Modula-2 (Oberon) als einzige vorgesehen ist, ist der sog. *modulimport*, der in abb. 12 für das modul *Foo* verwendet wurde. Hierdurch wird das importierende modul in einer weise vorgerüstet, dass wir durch sog. *qualifizierte bezeichner* („qualified identifier") der form `Foo.f` auf die exportierten bezeichner zugreifen können. Damit werden einerseits namenskonflikte umgangen; wir können jetzt etwa `Foo.g` und `g` (das eine abkürzung für `Bar.g` ist!) nebeneinander verwenden. Ausserdem wird aber auch die möglichkeit zu missverständnissen beim lesen des moduls durch menschliche leser eingeschränkt, da an jeder verwendungsstelle klargemacht wird, woher der bezeichner kommt. Der preis dafür ist ein erhöhter schreibaufwand und ein längerer programmquelltext.

2.6 Keine softwaretechniksprache? Was nun?

Viele der heute beliebten programmiersprachen sind definitiv keine softwaretechniksprachen (C, C++, ...), und trotzdem müssen mit ihnen grosse programmsysteme hergestellt werden. Was kann das modulkonzept hierbei helfen bzw. wie sollen die in diesem abschnitt vorgestellten mechanismen dann dargestellt werden?

Hierzu ist zunächst zu bemerken, dass — völlig unabhängig von der programmiersprache — die schaffung von modulen im hier definierten sinne mit klar definierten schnittstellen und einer rigorosen einhaltung des geheimnisprinzips immer möglich und auch in hohem mass ratsam ist. Wenn die programmiersprache die beschreibung von schnittstellen nicht ermöglicht, dann können diese trotzdem in dokumenten ausserhalb der programmiersprache beschrieben werden, besser noch: innerhalb von programmen in form von kommentaren, die u. u. eine standardisierte form haben, damit sie von werkzeugen behandelt werden können. Und wenn die programmiersprache die einhaltung einer abstraktion nicht erzwingt, dann kann diese dennoch durch textliche inspektion der programmquellen (durch werkzeuge, durch den programmierer selbst oder durch kollegen) überprüft werden.

Manche programmiersprachen (z. b. C) sind von einer ausgeprägten liberalität geprägt und lassen dem programmierguru nur ungern über den compiler kritik an seinen programmen ausrichten. Es gibt dann aber in der regel separate softwarewerkzeuge (z.B. `lint`), welche zusätzliche prüfungen anbieten.

Diese sollten genutzt werden!

Die folgenden regeln werden für den fall empfohlen, dass keine softwaretechniksprache zur verfügung steht und keins der angesprochenen zusätzlichen werkzeuge eingesetzt werden kann:

1. Sei möglichst spezifisch in der notation der schnittstellen (*header files*); mache keinen gebrauch von erleichterungsmöglichkeiten der programmiersprache (z. b. unvollständige funktionsprototypen).
2. Verwende niemals eine funktion mit unterschiedlichen anzahlen von parametern, auch nicht, wenn die programmiersprache dies ermöglicht.
3. Prüfe die vollständigkeit der implementierung gegen die schnittstellenbeschreibung: Sind alle dort erwähnten funktionen implementiert[9]? Haben sie den richtigen typ? Sind noch funktionen exportiert, die in der schnittstellenbeschreibung nicht vorkommen?
4. Prüfe die verwendungsstellen externer symbole gegen die schnittstellenbeschreibung: Stimmen die typen überein?
5. Schaffe getrennte namensräume, mindestens durch die etablierung und einhaltung geeigneter programmiernormen.
6. Schreibe spezifikationen im sinne des *design by contract* in form von standardisierten kommentaren (ähnlich JML) in die funktionsköpfe.

2.7 Was ist mit Java?

Die programmiersprache Java, die von vielen als das „bessere C++" angesehen wird und als sehr modern und fortschrittlich gilt, scheint softwaretechnikkonzepte bereits weitgehend zu unterstützen. Sie verfügt über eine relativ gute datenkapselung (mit abgestuften sichtbarkeitsregeln, s. u.) und kennt sogar eine `import`-anweisung. Trotzdem ist vorsicht geboten. Für Java-programmierer möchten wir die folgenden hinweise bezüglich des modulkonzepts geben:

Als grundlegende modulare einheit muss in Java das *paket* (`package`) gelten; die konstrukte, die hier exportiert werden können, sind klassen.

Es gibt drei methoden, klassen aus einem paket zu verwenden:

1. Einfach durch angabe des voll qualifizierten namens an der verwendungsstelle:

[9] In Unix kann das kommando `nm` dazu verwendet werden, die in einem objektmodul oder ausführbaren programm enthaltenen symbole auszugeben. Diese liste kann mit den symbolen aus dem *header file* verglichen werden.

```
vendor.util.foopack.Bar x = ... ;
```

Problematisch dabei ist, dass nirgendwo an zentraler stelle des programms vermerkt sein muss, dass dieses das paket `vendor.util.foopack` verwendet. Die tatsächlichen importe müssen deshalb durch textliche inspektion des gesamten programmtexts aufgesucht werden.

2. Durch spezifischen import einer klasse

```
import vendor.util.foopack.Bar;

Bar x = ... ;
```

Dies ist die einzige form des imports, welche den ansprüchen der softwaretechnik genügt. Kritisch anzumerken bleibt jedoch, was auch beim **FROM** x **IMPORT** y in Modula-2 angemerkt wurde, dass nämlich jegliche solche abkürzung für den programmierer immer wieder die möglichkeit zu missverständnissen und verwechslungen bietet. (Sog. IDE's, *integrated software development environments*, wie etwa Eclipse mildern dieses Problem ab, indem sie die entsprechenen Informationen einblenden, sobald sich die Maus über einen solchen Bezeichner bewegt.)

3. Durch pauschalen import

```
import vendor.util.foopack.*;

Bar x = ... ;
```

Dies ist die problematischste form der verwendung. Hier ist zunächst einmal völlig unklar, *welche* symbole durch diesen import in den namensraum des gegenwärtigen programms gelangen; das kann nur durch eine inspektion der schnittstelle von `vendor.util.foopack` geklärt werden. Importiert dieses paket allerdings — was leider häufig zu befürchten ist — andere pakete auf die gleiche weise, so tritt diese frage rekursiv erneut auf. Andererseits ist an der verwendungsstelle des solchermassen „heimlich" importierten symbols `Bar` völlig unklar, dass dieses genau aus diesem paket stammt. Es könnte ebensogut im laufenden programm deklariert oder aus einem anderen paket importiert sein.

Kontrollfragen

1. Was versteht man unter einem modul?
2. Nennen sie die drei aspekte, unter denen eine vernünftige modularisierung hilfreich ist!

3. Worin besteht das geheimnisprinzip?
4. Nennen sie die drei grundeigenschaften der Parnas'schen idee des geheimnisprinzips!
5. Wieso kann es sinnvoll sein, sich auf funktionale schnittstellen zu beschränken?
6. Was versteht man unter einer softwaretechniksprache?
7. Beschreiben sie die strategie des „design by contract"! Wieso ist es sinnvoll, die einhaltung der vorbedingungen an der aufrufstelle zu überprüfen?
8. Nennen sie drei aspekte, unter denen bei einer softwaretechniksprache die korrekte verwendung einer schnittstelle überprüft wird!
9. Auf welche weise garantiert Modula-2 die konsistenz zwischen schnittstellenbeschreibung und implementierung von modulen?
10. Erläutern sie den unterschied zwischen qualifizierendem import und modulimport!
11. Vergleichen sie die sichtbarkeitsregeln blockstrukturierter sprachen mit denen des modulkonzepts!

3 Systementwurf

Das kapitel über das modulkonzept hat natürlich die frage aufgeworfen, wie man denn zu einer modulstruktur kommen soll bzw. kann und wie man es anstellen kann, dass diese struktur möglichst „gut“ ist. Einen ersten ansatz dazu haben wir bereits präsentiert: die definition geeigneter datenabstraktionen. Mehr soll in diesem kapitel vorgestellt werden.

Eine gewisse schwierigkeit besteht darin, dass aus gründen der übungsorganisation wichtige phasen im prozess der softwarekonstruktion noch gar nicht vorgestellt worden sind und auch jetzt noch nicht dargestellt werden können.

3.1 Der systembegriff

Es geht in der softwaretechnik immer darum, ein *system* zu entwickeln, welches ein bestimmtes *problem* der praxis löst bzw. ein in der praxis schon existierendes system mit dem computer unterstützen soll. Auch das problem bzw. das existierende system wird eine bestimmte *struktur* haben, und es gibt grund zu der annahme, dass die struktur des zu bauenden systems dann besonders gut ist, wenn sie *problemorientiert* ist, d.h. der struktur des problems möglichst nahekommt. Dies erleichtert das finden einer systemstruktur und sorgt dafür, dass änderungen im problem zu leichter lokalisierbaren änderungen in der implementierung führen. Mit „möglichst nahe“ ist dabei gemeint, dass keine willkürlichen, unmotivierten abweichungen von der problemstruktur vorkommen sollen. Es wird ohnehin genügend gründe geben, in der implementierung von der problemstruktur abweichen zu müssen.

Da der systemgedanke eine wesentliche grundlage für den systementwurf darstellt, beginnen wir diesen abschnitt mit einer definition des begriffs *System* aus der skandinavischen informatiktradition (Holbæk-Hanssen u. a. 1977), aus der auch die objektorientierte programmierung (kap. 4) hervorgegangen ist:

Definition Ein system ist ein teil der welt, der von einer person oder einer gruppe von personen während eines bestimmten zeitraums und zu einem bestimmten zweck als eine aus komponenten gebildete einheit betrachtet wird. Dabei wird jede komponente durch merkmale beschrieben, die als relevant ausgewählt werden und durch aktionen, die in beziehung zu diesen merkmalen und zu den merkmalen anderer komponenten stehen.

Eine dekomposition eines komplexen systems in komponenten findet normalerweise schon bei einer voraufgehenden phase, der *problemanalyse* statt, wird aber dort nach rein logischen gesichtspunkten durchgeführt, die nur darauf abzielen, die verständlichkeit zu fördern.

In der analysephase wird das zu entwickelnde system also aus der sicht des *benutzers* (oder auch: der real existierenden welt) beschrieben. Der vorherrschende gedanke dabei ist, jeweils nur zu beschreiben, *was* zu tun ist, aber niemals, *wie* dies getan werden soll. Die strukturierung entspricht den anforderungen der anwendung.

Das implementierte system muss nach ganz anderen gesichtspunkten strukturiert sein. In der entwurfsphase muss das system nämlich aus der sicht der *maschine* beschrieben werden. Ein gängiges paradigma hierbei ist die bildung sog. *abstrakter maschinen*, welche stufenweise von den fähigkeiten der realen maschine abstrahieren und zu den anforderungen der anwendung hinleiten. Systementwurf beschäftigt sich sowohl mit dem leistungsumfang des systems („was?") als auch mit der realisierung („wie?"). Gegenüber der analyse findet hier also ein wesentlicher standpunktwechsel statt.

Aus der oben aufgeführten systemdefinition sollen einige dinge hier noch weiter ausgeführt werden:

Schnittstellen Aus der formulierung „ein teil der welt" ergibt sich, dass ein jedes system über bestimmte *schnittstellen* zum rest der welt (, der hier also ausserhalb des systems gesehen wird) verfügt. Je nachdem, ob über diese schnittstellen informationen aus der restwelt in das system hineinfliessen oder ob sie aus dem system in die restwelt hinausfliessen, sind dies eingabe- oder ausgabeschnittstellen. Dieser schnittstellenbegriff unterscheidet sich von dem begriff der modulschnittstellen.

Zustand Das system hat einen inneren zustand, der normalerweise nicht direkt beobachtbar ist (geheimnisprinzip!). Das system wird somit meist als *black box* verstanden, die „von aussen" undurchsichtig ist. Durch die wirkung von eingaben kann sich dieser zustand ändern; das system wird also auf gleiche eingaben nicht immer mit den gleichen ausgaben reagieren. In der o.a. systemdefinition ist dieser zustand durch das wort „merkmale" angesprochen; auch der begriff „attribute" ist gebräuchlich.

Aktionen Komponenten eines systems können aktionen ausführen (funktionen, methoden); dadurch kann sich der zustand dieser oder anderer komponenten ändern.

Verhalten Über die zeit hinweg zeigt ein system ein gewisses *verhalten*, wie es nämlich auf folgen von eingaben mit folgen von ausgaben reagiert. Dieses systemverhalten lässt unter umständen für einen beobachter der *black box* die bildung von theorien über den inneren zustand zu.

3.2 Systemarchitektur

Aufgabe des systementwurfs ist die herstellung einer *architektur des systems*, das bedeutet hier im wesentlichen eines systems von modulen inklusive der zwischen ihnen bestehenden benutzungs- bzw. aufrufstrukturen. Im idealfall enthält die systemarchitekturbeschreibung alle informationen, die der entwerfer für die weitere arbeit benötigt, so dass ein rückgriff auf dokumente der systemanalyse nicht mehr notwendig wird. Die modulstruktur ist von besonderer wichtigkeit für den fortgang des prozesses, da sie auch auswirkungen auf die organisationsstruktur des programmierteams hat und den grad der während der implementierung möglichen aufgabenteilung (parallelarbeit). Systementwurf ist ein kreativer prozess, der viel erfahrung und sicherlich auch begabung bei den entwicklern voraussetzt.

Von besonderer bedeutung in diesem zusammenhang ist der begriff der *entwurfsentscheidung*. Während der arbeit an einem system werden immer wieder entscheidungen getroffen, *wie* eine bestimmte datenstruktur denn nun im rechner dargestellt werden soll oder *warum* ein bestimmter algorithmus für den gegebenen zweck als geeignet ausgewählt und *wie* er dann in die programmiersprache umgesetzt wird. „Entscheidung“ bedeutet immer eine wahl zwischen mehreren alternativen, und häufig ist diese wahl im ersten ansatz falsch bzw. es stellt sich während der weiteren arbeit, die durch die alten entwurfsentscheidungen massiv beeinflusst wird, heraus, dass eine andere entscheidung besser gewesen wäre. Deshalb ist es sinnvoll, dass jede entwurfsentscheidung durch ein dazu eigens angelegtes modul eingekapselt wird, so dass man über die konsistent eingehaltene schnittstelle dieses moduls in verbindung mit dem geheimnisprinzip immer die möglichkeit hat, später die entwurfsentscheidung nochmals anders zu treffen.

Sehr häufig findet man in der diskussion um prinzipien des architekturentwurfs die begriffe *top down* und *bottom up*. Dabei handelt es sich jedoch eher um *beschreibungsansätze* für komplexe systeme als um echte *vorgehensmodelle*:

top down Die systembeschreibung geht von der logisch höchsten hierarchiestufe aus und verfeinert diese dann nach und nach bis zu den niedrigsten stufen. Für den leser hat dies den vorteil, dass er als erstes einen überblick über das gesamtsystem bekommt und anschliessend schrittweise so tief in das system eindringen kann, wie er es wünscht.

bottom up Die systembeschreibung beginnt auf der logisch niedrigsten hierarchiestufe, beschreibt also die bausteine des systems zuerst und dann die art und weise, wie diese schrittweise zu immer komplexeren einheiten zusammengesetzt werden, bis schliesslich das gesamtsystem beschrieben ist.

Die erfahrung hat gezeigt, dass systembeschreibungen nach der *top down*-methode besser verständlich sind. Häufig entsteht aber auch der eindruck, dass das system auf diese weise *konstruiert* worden ist, nämlich ausgehend von dem gesamtsystem und von dort durch stetige zerlegung bis zu den elementaren komponenten. Auch das zitat von Descartes im anhang E scheint diese strategie nahezulegen. In wirklichkeit lässt sie sich (ebenso wie die analoge *bottom up*-methode, bei der schrittweise immer komplexere komponenten aufgebaut würden, bis „zufällig" das fertige system herauskäme) nicht immer in reinform anwenden, auch wenn manche autoren (z. B. Klaus Wirth (1978)) das „Programmieren durch schrittweise Verfeinerung" als vernünftige methode zum algorithmenentwurf propagieren. Den wechsel zwischen *top down*- und *bottom up*-betrachtungsweise schildert schon Descartes (1637) in seiner schrift „Diskussion über die methode, seinen verstand richtig zu lenken und die wahrheit in den wissenschaften zu suchen" wie folgt:

Der forscher soll

- nur dasjenige als wahr annehmen, was der vernunft so klar ist, dass jeglicher zweifel ausgeschlossen bleibt,
- grössere probleme in kleinere aufspalten,
- immer vom einfachen zum zusammengesetzten hin argumentieren und
- das werk einer abschliessenden prüfung unterwerfen.

3.3 Modularisierung durch datenabstraktion

Das konzept des abstrakten datentyps ist von fundamentaler bedeutung für die gesamte informatik; schon in der Informatik-I-vorlesung (Klaeren (1990); Klaeren und Sperber (2007)) wird dies deshalb ganz klar dargestellt. Die erfahrung der programmierer besagt, dass datenstrukturen ein ganzes programm derart festzurren können, dass kaum noch eine bewegung möglich ist. Alan Perlis (1982) sagt deshalb in seiner sammlung von sinnsprüchen über die programmierung schon an zweiter stelle (von 130): *„Functions delay binding: data structures induce binding. Moral: Structure data late in the programming process."* Die erste programmiersprache, welche eine datenabstraktion in recht vernünftigem masse erlaubte, war Simula-67; dort hiessen die abstrakten datentypen „klassen". Simula-67 hat damit auch die objektorientierte programmierung begründet, und vieles von dem, was heute als ganz modern und neu gilt in der OO-programmierung, ist nur eine wiederentdeckung der abstrakten datentypen. Als reaktion auf die datenabstraktionsmöglichkeiten in Simula-67 hat sich in den siebziger jahren eine recht ausgefeilte theorie der abstrakten datentypen

entwickelt, die unter anderem auch algebraisch spezifizierte datentypen (Klaeren (1983); Ehrich u. a. (1989)) betrachtet hat. Bei diesem ansatz werden die relevanten eigenschaften der operationen eines abstrakten datentyps nur durch eine menge von gleichungen beschrieben, ohne irgendeine darstellung dieser operationen, geschweige denn der zugrundeliegenden datenstruktur anzugeben.

```
datatype Stack(item);

sorts stack, item;

constructors
          Clear:  → stack;
          Push : stack × item → stack;
operations
          Top : stack → item;
          Pop : stack → stack;
          IsEmpty: stack → Boolean;
          IsFull: stack → Boolean;

equations
          Top(Push(s,i)) = i;

          Pop(Push(s,i)) = s;

          IsEmpty(Clear)     = True;
          IsEmpty(Push(s,i)) = False;
end
```

Abbildung 13: Algebraische spezifikation eines abstrakten datentypen

Als beispiel betrachten wir den ADT „kellerspeicher“, für den es zahlreiche verwendungen und unterschiedliche implementierungen gibt. Es sollte keine rolle spielen, welche art von informationen im keller gespeichert wird; dies stellt sozusagen einen parameter für den ADT dar. Will man die grösste allgemeinheit der implementierung, wird man vermutlich eine verzeigerte liste bevorzugen: Ein eintrag im keller ist dann immer ein paar, bestehend aus einem element und einem zeiger auf den restlichen keller. Beim leeren Keller ist dies der nullzeiger und das element ist undefiniert. Zum einspeichern eines neuen elements wird platz alloziert, der alte zeiger in das element eingespeichert und der kellerzeiger auf das neue element umgestellt. Das top-element des kellers findet sich immer unter dem kellerzeiger, und das entfernen des obersten elements wird durch ersetzen des kellerzeigers durch den unter ihm

zu findenden restzeiger bewerkstelligt. Diese implementierung ist allerdings nicht sonderlich effizient, weswegen normalerweise die darstellung durch ein feld mit index bevorzugt wird. Dann hat der kellerspeicher aber eine maximale kapazität, die nicht überschritten werden darf[10].

Eine von jeder darstellung freie *algebraische spezifikation* des kellerspeichers findet sich in abb. 13. Das prädikat `isFull` scheint hier überflüssig zu sein, da es für die algebraische spezifikation keine rolle spielt, es ist jedoch ein vorblick auf die implementierungen, in der kellerspeicher begrenzt sind. Um solche implementierungen mit beschränkter kellertiefe nicht auszuschliessen, wurden hier keine gleichungen für `isFull` angegeben.

Abb. 14 zeigt den gleichen ADT in einer JML-Variante analog zu abb. 9. Wie dort legt auch hier die JML-spezifikation eines modells nicht notwendig eine implementierung durch ein feld nahe. Im vergleich zur algebraischen spezifikation zeigt sich ausserdem, dass das primäre argument vom typ `Stack` natürlich in den methodenköpfen nicht mehr auftaucht, da es in der objektorientierten programmierung anders übergeben wird. Um die verlegenheit, dass es in Java (vor version 1.5) keine parametrisierten klassen gibt, haben wir uns hier herumgemogelt, indem wir eine abstrakte klasse `Item` für die einzuspeichernden elemente vorausgesetzt haben.

Abstrakte datentypen als mengen zusammen mit spezifischen operationen liefern einen entscheidenden schlüssel zu der frage, woher module kommen bzw. wie man die „richtigen" module finden kann. Kernfragen beim systementwurf sind nämlich die folgenden:

1. Wie sehen die eingaben für das system aus?
2. Welche ausgaben soll das system erzeugen?
3. Welche datenstrukturen werden als grundlage eines algorithmus benötigt, um eingabe und resultate im programm abzulegen?
4. Welche operationen lassen sich auf diesen strukturen durchführen?
5. Welche davon sind weitgehend unabhängig voneinander? (d.h. für welche müssen keine gemeinsamen invarianten unterhalten werden?)

Voneinander abhängige datenstrukturen/typen und deren operationen kommen dann jeweils zusammen in je ein modul.

Damit hat man dann schon meistens 50-80% der module gefunden und einen eleganten entwurf dazu. Modulentwurf muss datenorientiert vorgehen.

[10] Übrigens eins der beliebten einfallstore für viren und andere schadsoftware.

```
public abstract class Stack {
   //@ public model Item [] stack;

   //@ public int index, capacity;

   /*@ ensures index == -1;
     @*/
   public abstract void clear();

   /*@ requires ! IsFull();
     @ ensures (index == \old(index) + 1)
     @         && (stack[index] == i);
     @         && (\forall int j;
     @                     0 <= j && j <= \old(index);
     @                     stack[j] == \old(stack[j]));
     @*/
   public abstract void push(Item i);

   /*@ requires ! IsEmpty();
     @ ensures \result == stack[index];
     @*/
   public abstract Item top();

   /*@ requires ! IsEmpty();
     @ ensures (index == \old(index) - 1)
     @         && (\forall int j;
     @                     0 <= j && j <= index;
     @                     stack[j] == \old(stack[j]));
     @*/
   public abstract void pop();

   /*@ ensures \result ==> (index == -1);
     @*/
   public boolean IsEmpty();

   /*@ ensures \result ==> (index == capacity);
     @*/
   public boolean IsFull();
}
```

Abbildung 14: ADT kellerspeicher in JML

3.4 Empfehlungen

Unabhängig von der methode, wie wir zu einer modulstruktur kommen, kann man die folgenden empfehlungen für den entwurf geben. Die meisten davon sind relativ „weich“ formuliert, da in der praxis viele kompromisse geschlossen werden müssen:

1. Die wichtigste empfehlung ist, das *geheimnisprinzip* wirklich ernstzunehmen und nach der devise des *design by contract* zu verfahren (s. abschn. 2.4).
2. Jede *entwurfsentscheidung* sollte in nur einem modul verkörpert sein. Dabei sprechen wir von einer entwurfsentscheidung, wenn irgend ein vorher unspezifiziertes detail (algorithmus, datenstruktur, ...) näher spezifiziert wird.
3. Jedes modul sollte mindestens eine entwurfsentscheidung beinhalten.
4. Jede entwurfsentscheidung soll mit ihrer begründung sorgfältig dokumentiert werden, da man sie vermutlich immer wieder in frage stellen wird.
5. Schnittstellen sollen *möglichst klein* sein, sowohl auf funktions- als auch auf modulebene. Bei modulen mit zu vielen funktionen stellt sich die frage, ob man sie nicht aufspalten sollte; funktionen mit zu vielen parametern sind ebenfalls problematisch[11].
6. Die modulstruktur (d.h. die importe zwischen den modulen), sollte unter allen umständen hierarchisch sein („DAG“, s. abb. 15), nicht zyklisch wie in abb. 16. Zyklische aufrufe sind nur innerhalb eines moduls zulässig.

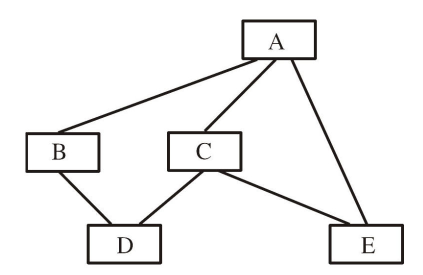


Abbildung 15: Zulässige modulstruktur

[11] (Perlis 1982, Nr.11): „Wenn du eine prozedur mit 10 parametern hast, dann hast du wahrscheinlich welche vergessen.“

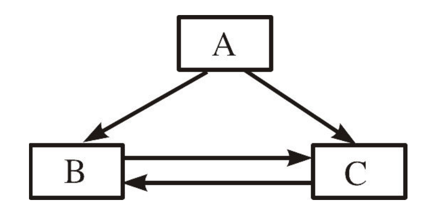


Abbildung 16: Zyklische modulstruktur

Das verbot zyklischer importe ist natürlich eine relativ strenge massnahme, welche die absicht verfolgt, alle eventuellen probleme zu vermeiden. Die in abb. 16 vorgeführte situation kann in der tat auch eine harmlose interpretation haben. Wenn die funktion B.f eine funktion C.g benötigt, aber C.g keine anderen funktionen aus B braucht und umgekehrt eine andere funktion C.h eine funktion B.i aufruft, aber B.i keine anderen funktionen aus C benötigt, so muss man zwar zu wartungszwecken unter umständen beide modultexte parallel bearbeiten, es treten jedoch keine grundsätzlichen verständnisprobleme auf. Haben wir dagegen eine situation, in der eine funktion B.f eine funktion C.g braucht, welche ihrerseits eine funktion aus B aufruft, welche möglicherweise wiederum eine funktion aus C aufruft, so können die module nicht mehr separat voneinander verstanden werden.

3.5 Qualität eines entwurfs

Zur beurteilung der qualität einer modulstruktur bieten sich folgende massstäbe an:

1. *Minimalität der schnittstellen*: Wird alles, was irgendwo exportiert wird, auch woanders importiert? Werden alle parameter von funktionen auch verwendet? (Dieses kriterium lässt sich z.t. syntaktisch durch werkzeuge überprüfen.)

2. *Ausgewogenheit gewisser masse*:

    - *Modulgrösse/Modulzahl*. Modulgrösse und modulzahl sind offensichtlich umgekehrt proportional zueinander; mit der anzahl der module steigen auch die kosten für die schnittstellenbeschreibung linear an. Auf der anderen seite wird jedes einzelne modul übersichtlicher, je kleiner es ist. In abhängigkeit von dem zugrundeliegenden problem gibt es immer einen optimalen bereich für das verhältnis von modulgrösse zu modulzahl.

- Importzahl (fan out)
- Verwendungszahl (fan in)
- Tiefe/Breite der modulstruktur. (Tiefe und breite der modulstrukturen sollen in vernünftiger relation zueinander stehen, s. abb. 17. Extremfälle einer sehr breiten und flachen wie auch einer sehr engen und tiefen modulstruktur bedürften einer besonderen legitimation.)

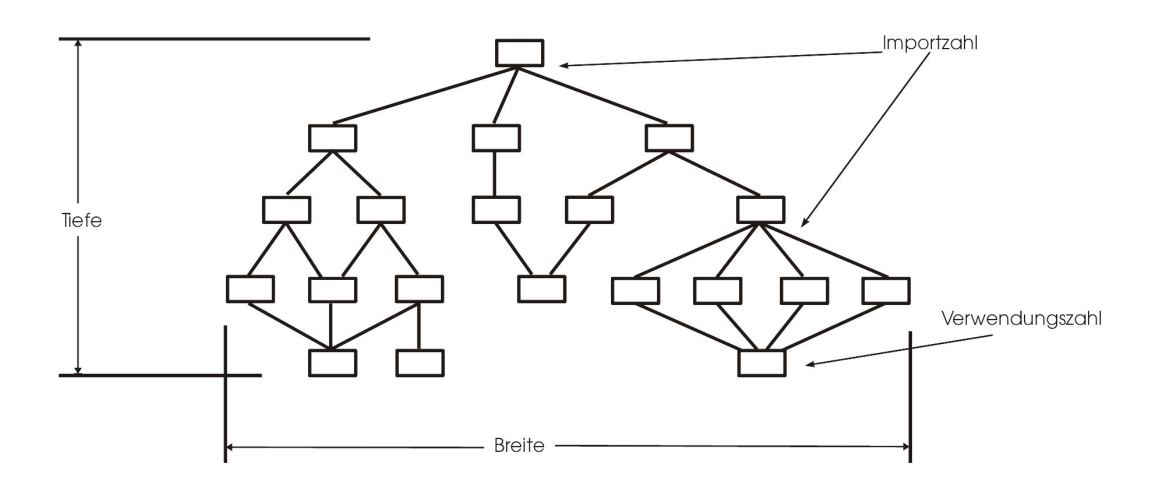


Abbildung 17: Tiefe und breite einer modulstruktur

3. *Modulbindung („cohesion")*: Modulbindung ist ein mass dafür, wie stark der zusammenhang zwischen den einzelnen prozeduren (und daten) eines moduls wirklich ist, d. h. dafür, ob das modul eine in sich abgeschlossene aufgabe realisiert. Modulbindung ist eine eigenschaft jedes einzelnen moduls.

   Zufällige bindung liegt vor, wenn kein prinzip erkennbar ist, nach dem prozeduren bzw. daten in das modul aufgenommen wurden.

   Logische bindung liegt vor, wenn alle prozeduren sich unter einem logischen gesichtspunkt zusammenfassen lassen; z.b. alle funktionen zur bedienung eines terminals etc.

   Temporale bindung liegt vor, wenn die gemeinsamkeit zwischen den prozeduren in einem modul darin besteht, dass sie alle zum (annähernd) gleichen zeitpunkt ausgeführt werden (z.b. div. initialisierungs- und finalisierungsfunktionen).

   Prozedurale bindung liegt vor, wenn die elemente eines moduls voneinander abhängig sind und in bestimmten reihenfolgen ausgeführt

werden müssen. Ein beispiel bildet etwa der in Pascal eingebaute ADT „`FILE`“, der dort allerdings kein modul darstellt, sondern in die programmiersprache eingebettet ist. Hier dürfen prozeduren nur in der in abb. 18 angegebenen reihenfolge aufgerufen werden.

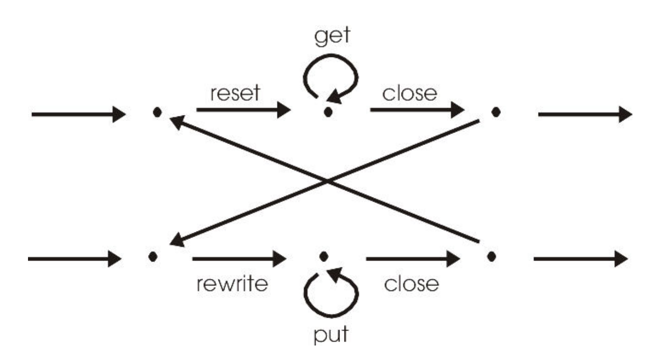


Abbildung 18: Prozedurale bindung für ADT FILE (Pascal)

Kommunikationsbindung liegt vor, wenn alle funktionen auf einer gemeinsamen datenstruktur operieren (z.b. abstrakter datentyp).

Sequentielle bindung liegt vor, wenn alle prozeduren nur in einer sequentiellen folge ausgeführt werden können und diese folge auch bis zum ende durchlaufen werden muss.

Funktionale bindung liegt vor, wenn ein modul überhaupt nur eine einzige prozedur exportiert.

4. *Modulkopplung („coupling“)*: Modulkopplung ist ein mass dafür, wie stark die einzelnen module eines systems voneinander abhängig sind. Kopplung ist eine eigenschaft zwischen je zwei modulen eines systems.

Datenkopplung liegt vor, wenn die argumentlisten der prozeduren des untergeordneten moduls nur primitive datenelemente beinhalten.

ADT-kopplung liegt vor, wenn die argumentlisten komplexe datenstrukturen beinhalten (und somit das untergeordnete modul diesen datentyp ebenfalls kennen muss).

Steuerkopplung liegt vor, wenn bestimmte argumente nur dazu verwendet werden, um im untergeordneten modul bestimmte pfade im steuerfluss auszuwählen (z.B. `OPEN (filename, false)` ergibt fehlermeldung, wenn `filename` nicht existiert; `OPEN (filename, true)` legt `filename` neu an, falls es noch nicht existiert.)

Externe kopplung liegt vor, wenn module über globale datenbereiche ausserhalb der modulstruktur kommunizieren.

Inhaltskopplung liegt vor, wenn das übergeordnete modul details der implementierung des untergeordneten moduls verwendet.

3.6 API-Entwurf

Henning (2007) weist darauf hin, wie wichtig entwurfsfragen gerade im zusammenhang mit APIs („Application Programmer's Interfaces“) sind: hier muss überlegt werden, in welchen zusammenhängen die funktionen einer bibliothek in der regel verwendet werden, so dass dem anwendungsprogrammierer das leben nicht unnötig erschwert wird. Als beispiel führt er die funktion `Select()` aus dem .NET-framework an, betont dabei aber, dass es ähnlich problematische beispiele überall gibt, und gewiss nicht nur bei Microsoft.

Die typische anwendung von `Select()` ist innerhalb der hauptschleife eines servers, der anfragen von mehreren klienten verarbeiten möchte: in jedem schleifendurchlauf werden alle die sockets bedient, die gerade bereit sind, dann wird `Select()` erneut aufgerufen:

```
public static void Select (List checkRead, List checkWrite,
                           List checkError, int microseconds);
// Server code
int timeout = ...;
ArrayList readList = ...; //Sockets to monitor for reading
ArrayList writeList = ...; //Sockets to monitor for writing
ArrayList errorList = ...; //Sockets to monitor for errors

while (!done) {
   SocketList readTmp = readList.Clone();
   SocketList writeTmp = writeList.Clone();
   SocketList errorTmp = errorList.Clone();
   Select(readTmp, writeTmp, errorTmp, timeout);
   for (int i = 0; i < readTmp.Count; ++i) {
      // deal with each socket ready for reading
   }
   for (int i = 0; i < writeTmp.Count; ++i) {
      // deal with each socket ready for writing
   }
   for (int i = 0; i < errorTmp.Count; ++i) {
      // deal with each socket with errors
   }
}
```

Das erste problem entsteht hier schon dadurch, dass `Select()` seine ersten drei argumente überschreibt. Deswegen müssen diese listen, auch wenn sie über längere zeiträume unverändert bleiben, vor jedem aufruf kopiert werden, was umständlich ist und nicht gut skaliert.

Ein zweites problem ist, dass die liste der sockets in der `errorList` in der regel die vereinigung der sockets der `readList` und der `errorList` ist. Wenn ein server also je 100 sockets für das lesen und schreiben überwachen soll, dann muss er in jedem schleifendurchlauf 400 listenelemente kopieren. Das dritte

problem ist, dass zwar (als viertes argument) ein timeout in mikrosekunden angegeben wird, aber dass `Select()` den rückgabetyp `void` hat, also nicht gesondert mitteilt, ob überhaupt irgendeiner der sockets bereit gewesen ist. Dem anwendungsprogrammierer bleibt also nichts anderes übrig, als die länge aller drei listen zu überprüfen, um festzustellen, ob überhaupt irgendetwas zu tun ist. Schlimmer noch: auch wenn gar nichts auf irgendeinem der sockets passiert ist, hat `Select()` die argumentlisten zerstört, so dass sie erneut kopiert werden müssen. Eine sinnvollere schnittstelle für `Select()` wäre die folgende gewesen:

```
public static int Select(ISet checkRead,
                         ISet checkWrite,
                         Timespan seconds,
                         out ISet readable,
                         out ISet writeable,
                         out ISet error);
```

Offensichtlich hat der entwerfer der schnittstelle in .NET nicht darauf rücksicht genommen, wie diese funktion denn später verwendet werden soll.

Konkret gibt Henning unter anderem die folgenden hinweise für die entwicklung von APIs:

- Ein API sollte möglichst klein sein. Je weniger typen, funktionen und parameter vorkommen, desto leichter ist das erlernen, erinnern und korrekte benutzen. Als gegenbeispiel führt er die string-klasse von C++ an, die mehr als hundert funktionen beinhaltet und ohne handbuch praktisch nicht zu benutzen ist. Ein anderes beispiel gibt es im Unix-kernel: von den funktionen `wait()`, `waitpid()`, `wait3()` und `wait4()` wäre im grunde nur `wait4()` notwendig, weil sich die anderen drei damit realisieren lassen. Der programmierer muss nun aber die dokumentation zu allen vier funktionen lesen, bis er weiss, welche davon er verwenden soll.

- APIs können nur entworfen werden, wenn ihr einsatzkontext verstanden ist. Als beispiel führt er für die überall vorkommenden *name-value pairs* die funktion `lookup()` an: was soll diese funktion tun, wenn zu dem nachgefragten namen gar kein wert gespeichert ist? Hier bieten sich drei möglichkeiten an:

  - eine exception werfen,
  - einen *null*-zeiger zurückgeben oder
  - einen leeren string zurückgeben.

  Die entscheidung zwischen diesen alternativen hängt davon ab, in welchem kontext die tabelle verwendet werden soll: Wenn es als fehler betrachtet wird, einen nicht-existenten eintrag abzufragen, ist die exception

sicher die richtige lösung. Wenn dies nicht so ist und wenn es wichtig ist, einen nicht-existenten eintrag von einem ausdrücklich mit dem leeren string versehenen eintrag zu unterscheiden, muss der *null*-zeiger zurückgegeben werden. Ist dies nicht wichtig, kann auch der leere string zurückgegeben werden.

- APIs sollten aus der perspektive des aufrufers geschrieben werden, damit ihm die arbeit erleichtert wird. Als beispiel dient hier eine funktion zur erzeugung der schnittstelle für einen fernseher. Ist die funktion deklariert als

```
void makeTV(bool isBlackAndWhite,
            bool isFlatScreen) {
      ...
      }
```

  so kann der anwendungsprogrammierer im aufruf `makeTV(false,true)` ohne blick in die dokumentation nicht erkennen, welche wirkung die parameter haben. Besser wäre eine deklaration der form

```
enum ColorType {Color, BlackAndWhite};
enum ScreenType {CRT, FlatScreen};
void makeTV(ColorType col, ScreenType st);
```

  Dann kann der o. a. aufruf als `makeTV(Color,FlatScreen)` geschrieben werden und ist somit selbsterklärend.

- APIs sollten dokumentiert werden, bevor sie implementiert werden. Die gefahr der dokumentation nach der implementierung ist, dass der implementierer „mental verseucht" (orig.: *mentally contaminated*) durch seine implementierung ist und dann beschreibt, was er getan hat, und nicht, wie es benutzt werden soll. Dabei lässt er häufig dinge weg, die ihm aufgrund seiner implementierung hundertprozentig klar sind, die aber keineswegs selbstverständlich sind.

Ein gesellschaftliches problem, das Henning in diesem zusammenhang anspricht, ist die tradition, dass programmierer, die eine gewisse erfahrung erworben haben, normalerweise in höherwertige positionen befördert werden (z. B. projektmanager). Dann steht ihre erfahrung aber nur noch eingeschränkt zur verfügung und der meiste code wird von unerfahrenen neulingen geschrieben.

Kontrollfragen

1. Wiederholen und kommentieren sie die systemdefinition nach Holbæk-Hanssen u. a. (1977)!

2. Welche details gehören zur schnittstellenbeschreibung einer funktion? Welche rolle spielen globale variablen dabei?
3. Nennen sie maßstäbe zur beurteilung der qualität einer modulstruktur!
4. Warum sind zyklische importe zwischen modulen problematisch?
5. Was versteht man unter „prozeduraler bindung“? Nennen sie ein beispiel!
6. Nennen sie gemeinsamkeiten und unterschiede zwischen abstrakten datentypen und modulen!

4 Objektkonzept, OOD und OOA

Wie bereits erwähnt, wurde die objektorientierte programmierung erfunden im zusammenhang mit der programmiersprache Simula 67. Ziel dieser sprachentwicklung war es, simulationen von systemen der realen welt zu beschreiben. Ein vielzitiertes beispiel hierzu ist etwa das *postamt*. Hier gibt es eine menge von *schaltern*, an denen bestimmte *dienste* angefordert werden können und eine anzahl von *kunden*, welche diese schalter besuchen wollen und sich dabei unter umständen in eine *warteschlange* einreihen müssen. Alle schalter haben gewisse *merkmale (attribute)* gemeinsam, zum beispiel, dass sie geöffnet oder geschlossen sein können oder eben dass sich vor ihnen warteschlangen bilden können. In anderen merkmalen unterscheiden sich die schalter; zum beispiel gibt es schalter, an denen briefmarken verkauft werden, es gibt postbankschalter, paketschalter, telefonschalter etc. Auch die kunden haben gewisse merkmale gemeinsam, in anderen unterscheiden sie sich. So gibt es etwa paketkunden, briefkunden, postbankkunden etc. Es ist auch nicht auszuschliessen, dass ein kunde mehrere geschäfte zu erledigen hat und so etwa von einem postbankkunden zu einem paketkunden wird oder umgekehrt. Implizit angesprochen wurde in dieser darstellung bereits, dass kunden und schalter(beamte) in der lage sind, bestimmte *operationen* auszuführen.

Eine simulation des postamts würde nun nach vorgegebenen wahrscheinlichkeitsverteilungen schalter öffnen und schliessen und kunden das postamt betreten lassen, um dort bestimmte geschäfte zu erledigen und es dann wieder zu verlassen. Zweck der übung wäre es vermutlich, bestimmte statistische daten zu sammeln wie etwa durchschnittliche und maximale verweildauer der kunden im postamt, länge der warteschlange, auslastungsgrad der schalter usw. Die macher von Simula 67 haben sich entschieden, konzepte der realen welt wie etwa hier schalter, warteschlangen, kunden als *objekte* zu bezeichnen und diese objekte in *klassen* zusammenzufassen, soweit sie gleichartig sind. Von klassen können *unterklassen* gebildet werden, welche automatisch alle eigenschaften der oberklasse *erben* und andere eigenschaften hinzufügen können, oder aber auch vorgegebene merkmale der oberklasse abändern. Es ergibt sich nahezu zwanglos, dass solche objekte während der laufzeit dynamisch erzeugt und wieder vernichtet werden können und dass sie in der zeit zwischen erzeugung und vernichtung ein eigenleben führen können, das heisst also, von sich aus (spontan) irgendwelche aktionen ausführen können. Schaut man eine ebene tiefer in die implementierung hinein, so ergibt sich, dass derartige objekte ansammlungen von *daten* und *prozeduren* darstellen. Die daten charakterisieren hierbei so etwas wie den inneren zustand des objekts, während die prozeduren die möglichen aktionen des objekts beschreiben.

Simula 67 verfügte über keine vorrichtungen zur datenkapselung (d.h. der

verhinderung illegaler zugriffe auf den internen zustand von objekten); in der tat entstanden die theoretischen arbeiten (charakteristisch z.B. Hoare (1972)) zum sinn der datenkapselung erst durch die anregung der programmierung in Simula 67. Heute gehört die datenkapselung zu den wichtigsten ingredienzen des objektorientierten ansatzes.

Simula 67 vermischte im übrigen das objektkonzept mit den hiervon völlig unabhängigen konzepten des zeigers und der nebenläufigen programmierung in form der sog. *coroutine*. Erst reichlich zehn jahre später hat die sprache Smalltalk die notwendigen konzepte der objektorientierten programmierung in einer sauber definierten minimalen form zusammengetragen. Smalltalk entstand unter leitung von Alan Kay am XEROX PARC, nachdem Kay die quellen eines Simula-compilers studiert hatte. Moderne ideen zur objektorientierten programmierung basieren in der regel auf dem Smalltalk-modell. Eine hervorstechende eigenschaft von Smalltalk ist, dass programme in einen *bytecode* übersetzt werden, der dann von einer *virtuellen maschine (VM)* interpretiert wird.

Leider ist die aktuelle literatur über die objektorientierte programmierung sehr stark von den vorherrschenden programmiersprachen bestimmt, und hier sogar noch besonders von den zweitklassigen OO-sprachen C++ und Java. Dadurch entsteht eine menge ganz spezifischer probleme, die den blick auf die wirklich objektorientierte sicht- und vorgehensweise verstellen:

- Die dynamische erzeugung und vernichtung von objekten wird vermischt mit fragen der speicherallokation und -deallokation. Natürlich ist es klar, dass auf der implementierungsseite auch speicher alloziert werden muss, wenn ein objekt erzeugt wird, und dass dieser speicher wieder freigegeben werden kann, wenn das objekt vernichtet wird, aber die objektorientierte programmierung sollte auf einer höheren ebene denken und sich um solche implementierungsdetails nicht kümmern müssen. Bezeichnend ist es ja bereits, dass die „konstruktoren“ in C++ und Java nicht wirklich objekte „konstruieren“, sondern sich lediglich um die initialisierung bereits allozierter objekte kümmern. Dementsprechend müssen die destruktoren alle komponenten des objekts (rekursiv) freigeben. Hier liegt ein wesentlicher entwurfsfehler von C++, wo der programmierer die destruktoren selbst programmieren muss und dabei selbstverständlich fehler machen kann. Dann entstehen sogenannte *memory leaks*, d. h. also stellen, wo nicht mehr über zeiger erreichbarer speicher belegt bleibt. Richtig wäre es gewesen, die sprache so zu entwerfen, dass automatische *speicherbereinigung* (*„garbage collection“*) möglich gewesen wäre. Hierfür ist C++ nicht geeignet, da wir dafür an jeder stelle von datenstrukturen in der lage sein müssten, zeiger von anderen daten zu unterscheiden, und dies ist bei C++ wegen des schwachen typsystems nicht möglich. Eine möglichkeit, dies

richtig zu machen, zeigt etwa Java.

- Bei der objektorientierten modellierung kommen ganz selbstverständlich *objektreferenzen* vor, d. h. also verweise von einem objekt auf ein anderes. Häufig werden diese referenzen mit zeigern gleichgesetzt, was ebenfalls zu kurz gedacht ist: Einerseits gibt es häufig *persistente objekte*, deren lebensdauer die programmausführung überschreitet (und die deshalb auf externen speichermedien abgelegt werden müssen, wo sie nicht durch zeiger ansprechbar sind. Andererseits hat gerade objektorientierte programmierung auch den gedanken des verteilten programmierens sehr stark propagiert; dann ist aber eine objektreferenz kein zeiger, sondern eher so etwas wie ein URI („Uniform Resource Identifier").

- Datentypen und klassen werden miteinander vermischt. Klassen können an ziemlich vielen stellen legal vorkommen, wo im grunde typen erwartet würden (z. b. bei variablendeklarationen und innerhalb von prozedurköpfen); trotzdem sind sie jedoch konzeptuell von diesen unterschieden. Sauber in dieser hinsicht ist nur Smalltalk, wo es typen im üblichen sinne gar nicht gibt, sondern nur klassen.

Dass sich das ältere Smalltalk nicht gegenüber den späteren, schlechteren OO-sprachen durchsetzen konnte, hat mehrere gründe:

- Smalltalk brauchte von anfang an grafische displays und war nicht auf effizienz getrimmt; deshalb lief es zunächst nur auf leistungsfähigen workstations, die entsprechend teuer und wenig verbreitet waren.

- Smalltalk kennt kein konzept einer datei bzw. eines dateisystems und wirkt deshalb fremdartig auf programmierer, die es gewohnt sind, mit dateien zu arbeiten. Alles in Smalltalk ist ein objekt, und die lebendigen objekte werden von der Smalltalk-VM in einem *Smalltalk image* aufbewahrt.

- C++ traf auf eine grosse zahl ausgebildeter C-programmierer, die „nur noch" die erweiterten konzepte lernen mussten

- Deshalb basiert auch Java, das von Smalltalk u. a. die idee einer virtuellen maschine übernahm, aus marketing-gründen lieber auf C++.

4.1 Klassen und objekte

Grady Booch definiert ein objekt als

> *„eine entität, die einen zustand hat, die charakterisiert ist durch die aktionen, die sie einerseits ausführt und die sie andererseits von anderen objekten anfordert. Ein objekt ist eine instanz einer klasse, wird bezeichnet durch einen namen, hat eine eingeschränkte sicht auf andere objekte und eine eingeschränkte sichtbarkeit für andere objekte."*

In der tat ist alles, was gegenüber den abstrakten datentypen hier an wesentlichem noch hinzukommt, durch die begriffe der *instanziierung* von objekten aus klassen und die *vererbung* von oberklassen auf unterklassen vollständig beschrieben. Alles übrige ist ein reiner wechsel der terminologie, zum beispiel, dass prozeduren jetzt *methoden* genannt werden und prozeduraufrufe *nachrichten* (manche autoren bevorzugen das wort *botschaften*).

Eine klasse stellt gewissermassen eine *fabrik für objekte* dar; sie ist wie ein stempel oder eine schablone, vor der immer neue abdrücke gezogen werden können. Die solchermassen dynamisch erzeugten objekte heissen dann auch *instanzen* (besser: *exemplare*) der klassen. Obwohl solche exemplare nach einem einheitlichen strickmuster von der klasse abgezogen werden, haben sie dennoch jedes eine eindeutige *objektidentität*; sie können von anderen exemplaren der gleichen klasse unterschieden werden. Zum zugriff auf objekte dient eine *objektreferenz*; zwei objekte haben dann die gleiche identität, wenn sie die gleiche objektreferenz haben. Objektreferenzen sind in der regel komplex strukturiert; sie mit zeigern zu verwechseln, ist, wie gesagt, ausgesprochen kurzsichtig.

4.2 Statische modellierung

In diesem zusammenhang muss die UML (*Unified Modeling Language*) (Fowler und Scott 1998; Störrle 2005) erwähnt werden, die durch vereinigung und vereinheitlichung mehrerer ansätze (von Booch, Jacobson und Rumbaugh) entstanden ist und auch von der OMG unterstützt wird. Wir versuchen uns in dieser vorlesung möglichst in der UML auszudrücken, auch wenn diese ein bewegliches ziel darstellt, da häufig neue versionen davon publiziert werden[12]. Der begriff „modeling *language*" ist eigentlich irreführend, denn es handelt sich nicht wirklich um eine *sprache*, sondern in der hauptsache um eine sammlung verschiedener graphischer notationsmittel. Diese werden wir in der vorlesung nicht alle ansprechen, sondern nur die wichtigsten davon.

Wie Bowen und Hinchey (2006) argumentieren, darf man sich nicht zu viel von UML versprechen: die bedeutung der zahlreichen formen von diagrammen ist bei weitem nicht formal genug definiert, und über die frage, zu welchem zweck welche diagramme zu verwenden sind, gibt es nicht sehr viel

[12] B. Meyer interpretierte die abkürzung als „Uninterrupted Marketing Literature".





Abbildung 19: Camel and horse

einigkeit. Als vereinigung („unified“!) von mehreren unterschiedlichen ansätzen, die zum teil recht alt sind (1970er jahre) ist UML ausserdem nicht sonderlich homogen; die begründung dafür, was in UML aufgenommen wurde und was aussen vor gelassen wurde, ist häufig nur in der durchsetzungskraft einzelner personen zu finden; die einzelnen ausdrucksmittel sind im übrigen nicht besonders gut aufeinander abgestimmt. Dies ist ein typisches phänomen im zusammenhang mit der arbeit von kommitees, siehe Abb. 19; oft zitiert wird der ausspruch: „A camel is a horse designed by a committee.“ Ein weiteres problem besteht darin, dass die objektorientierten programmiersprachen z. t. ganz divergierende vorstellungen von „objekt“, „klasse“, „sichtbarkeit“ und ähnlichen konzepten haben, und dass UML versucht, allen diesen sprachen entgegenzukommen.

Eine recht brauchbare und gut lesbare einführung in UML 2 mit sehr aussagekräftigen beispielen (aus einer fallstudie „fluglinie“) gibt Störrle (2005).

Bei der in diesem abschnitt angesprochenen *statischen modellierung* geht es

Konto
Nummer Besitzer Saldo
Einzahlen Abheben Abfragen

Abbildung 20: Klassennotation am beispiel einer kontoklasse

darum, die klassen eines objektorientierten modells zu beschreiben und die *beziehungen* zwischen ihnen. Dies geschieht hauptsächlich durch die *klassendiagramme*. In der aus guten gründen später stattfindenden *dynamischen modellierung* wird es dann um das verhalten von objekten über einen zeitverlauf hinweg gehen. Wie Störrle (2005) bemerkt, lassen sich unter dem begriff „klasse“ ganz unterschiedliche bedeutungen finden:

Konzept In der analysephase ist eine klasse oft nicht mehr als ein bestimmtes konzept, das sich beim analysieren des problems auffindet.

Objektmenge Oft werden klassen aber auch als mengen irgendwie gleichartiger objekte betrachtet, ohne dass deren verwendungsmöglichkeiten eine rolle spielen.

Typ Wie bereits oben angesprochen, werden klassen oft im sinne von (daten-) typen verwendet. Objekte sind dann werte dieser typen.

Implementierung Nicht zuletzt bezeichnet „klasse“ oft auch ganz einfach den programmtext einer klasse, z. b. in C++ oder Java.

Den prozess, wie man zu einem objektorientierten modell kommt, werden wir in abschn. 4.5 noch besprechen.

Klassen werden in form von kästen gezeichnet, die in vertikaler richtung dreigeteilt sein können. Im einfachsten fall enthält so ein kasten nur den namen einer klasse, in der ausführlichsten version steht im obersten teil der *name* der klasse, in der mitte die *attribute*, im untersten teil die *methoden*, s. abb. 20. Auch die zweigeteilte form (nur namen und attribute) kommt vor. Kommt es nur auf die beziehungen zwischen klassen an oder ist das bild recht komplex, so reduziert man die beschriftung eines solchen kastens auf den reinen namen der klasse.

Während der modellierung liegen viele attribute und assoziationen nur „im auge des betrachters“, d. h. über deren spätere entsprechungen in der imple-

mentation wird noch nichts impliziert. Es ist durchaus normal, zunächst nur namen von klassen zu erfassen und beziehungen zwischen ihnen; nach und nach kommen dann auch attribute und methoden hinzu.

Zwischen klassen können verschiedenartige beziehungen bestehen. Bei dem wort „beziehung“ ist es dabei durchaus richtig, an eine *relation* im sinne der mathematik zu denken.

4.2.1 Assoziation

Der allgemeinste typ von beziehungen (relationen) zwischen klassen heisst in der UML *assoziation*; die entstehende struktur heisst dementsprechend *assoziationsstruktur*. Assoziationen werden durch einfache linien zwischen den klassen notiert, wobei der name der beziehung (z. b. „ist vorgesetzter von“, „hört vorlesung“, „prüft“ etc.) an der linie steht. Dieser name der beziehung heisst in der UML-welt auch eine *rolle*, die dann auch unsymmetrisch bezeichnet werden kann, je nachdem, in welcher richtung die beziehung gelesen wird. Mit assoziationen sind häufig aussagen über die *kardinalitäten* der entsprechenden relationen verbunden; diese werden durch zahlen bzw. zahlenpaare an der verbindungslinie in der nähe der klasse notiert.

In abb. 21 (Fowler und Scott 1998) sehen wir beispielsweise eine beziehung zwischen personen und firmen. Eine firma kann zwischen `0` und beliebig vielen personen diese beziehung haben (`Person` in der rolle `employee`); umgekehrt kann eine person diese beziehung zu `0` bis beliebig vielen firmen haben (`Company` in der rolle `employer`). In dieser assoziation von klassen liegt ein sachverhalt, der für die modellierung eine besondere rolle spielt, weil hierzu weitere daten im system gespeichert werden müssen. Solche sachverhalte werden als separate klassen modelliert, die dann *assoziationsklassen* heissen und durch eine gestrichelte linie an die assoziationslinie angebunden werden. Im beispiel finden sich zwei solche assoziationsklassen mit ihren zugeordneten attributen: `Job` informiert zusätzlich zum sachverhalt einer assoziation zwischen einer `Person` und einer `Company` ausserdem noch über die beschreibung des jobs, das anfangsdatum und das gehalt. Die klasse `Marriage` stellt eine assoziation der klasse `Person` mit sich selbst dar, wobei eine der personen die rolle `husband` und die andere die rolle `wife` spielt. Zu einer ehe ist es interessant, datum und ort der eheschliessung zu speichern.

Häufig sind die assoziationen zwischen klassen noch durch besondere invarianten gekennzeichnet, die in der UML *einschränkungen (constraints)* heissen. Um einschränkungen zu notieren, können diese freitextlich in geschweiften klammern an die verbindungslinie geschrieben werden; zu bevorzugen ist jedoch eine notation in der eigens hierzu entworfenen *OCL (object Constraint Language)*. In abb. 21 könnte es z. b. interessant sein, festzuhalten, dass bei ei-

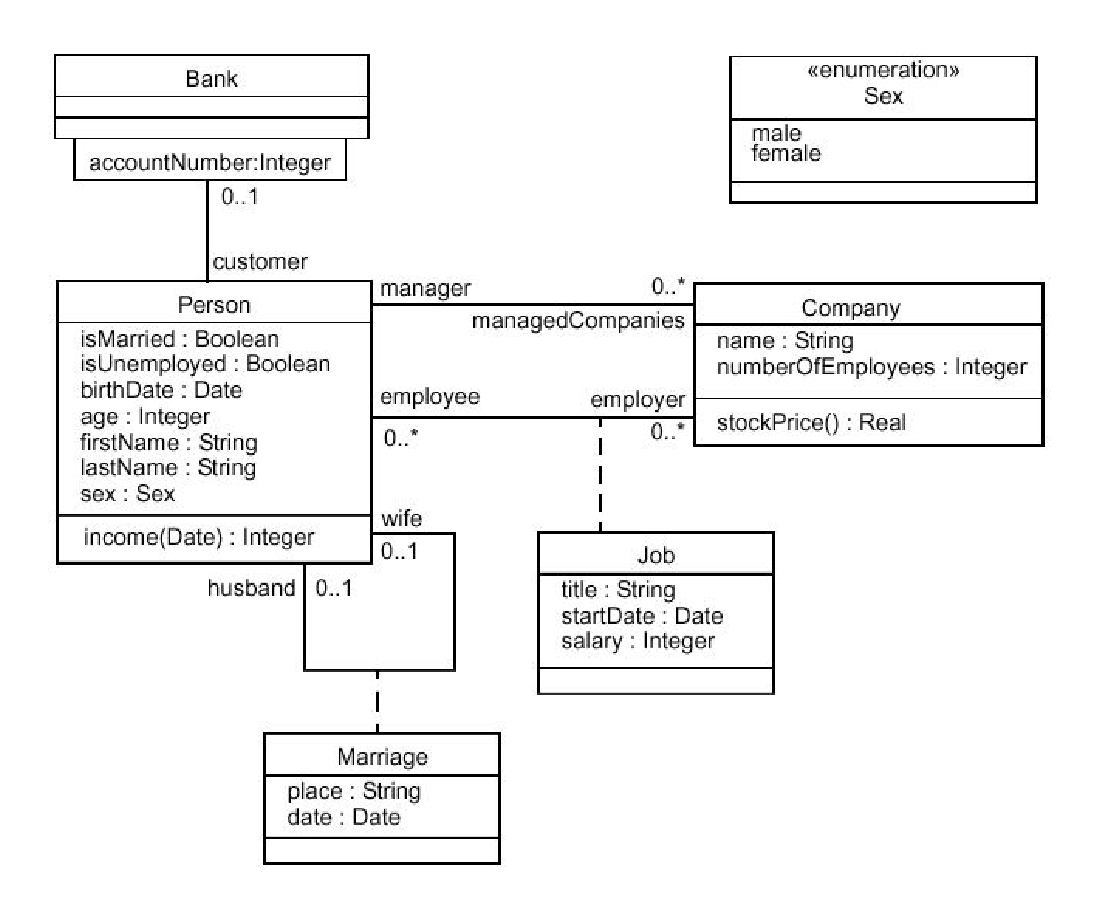


Abbildung 21: Assoziationsstruktur

ner ehe die ehefrau weiblich und der ehemann männlich und beide partner alt genug sind:

```
context Person inv:
   self.wife->notEmpty() implies (self.wife.sex = Sex::female
                         and self.wife.age >= 18)
   and self.husband->notEmpty() implies (self.husband.sex = Sex::male
                         and self.husband.age >= 18)
```

In der implementierung sind assoziationen häufig dadurch abgebildet, dass ein objekt eine objektreferenz („zeiger“) auf ein anderes objekt enthält. Es ist jedoch falsch, diesen gedanken bereits zu früh während der modellierung in den vordergrund zu rücken. Die UML erlaubt es, dies durch pfeilspitzen an den assoziationslinien anzudeuten; beispiele hierfür finden sich in abb. 23. Hierdurch wird impliziert, dass später im laufenden system ein objekt „weiss“, mit welchen anderen objekten es in dieser assoziation steht. Im klartext gesprochen heisst dies, dass dieses objekt die objektreferenzen derjenigen objekte besitzt, die mit ihm assoziiert sind.

4.2.2 Aggregation, Komposition

Eine klasse kann andere klassen *enthalten* bzw. *bestandteil* anderer klassen sein; je nachdem, von welcher seite wir diese relation betrachten, nennen wir sie auf englisch eine *has-a*-beziehung oder *part-of*-beziehung[13]. Wir sprechen hier auch von einer *aggregation* und nennen die möglicherweise komplexe aufbrechung einer klasse in eine hierarchie von teilklassen eine *aggregationsstruktur*. Aggregationen werden in der UML durch einen pfeil mit einer (weissen) raute an einem ende notiert, s. abb. 22. Die UML macht hier noch einen feinen unterschied (der letztlich vermutlich wieder aus einer vermischung von modellierungs- und speicherallokationskonzepten beruht): Besonders „starke“ formen der aggregation, bei denen die bestandteile eines exemplars zu existieren aufhören, wenn das exemplar der oberklasse vernichtet wird, heissen hier *kompositionen* und werden durch pfeile mit einer ausgefüllten raute dargestellt. Die bedeutung von abb. 22 ist also, dass ein polygonzug aus mindestens 3 bis beliebig vielen punkten besteht, die ausserdem angeordnet sind, dass ein kreis aus genau einem punkt (dem mittelpunkt) besteht, wobei mit der vernichtung eines polygonzugs bzw. eines kreises auch die zugehörigen punkte verloren gehen. Darüber hinaus haben polygonzüge und kreise auch noch einen darstellungsstil, der jedoch von diesen geometrischen objekten unabhängig ist und auch bei ihrer vernichtung erhalten bleibt. Der unterschied

[13]Mandeep Singh machte mich darauf aufmerksam, dass der *has-a*-begriff problematischer ist als der *part-of*-begriff: „Mike has a car“ bedeutet nicht, dass das auto ein bestandteil von Mike ist, deswegen handelt es sich hier nicht um eine *has-a*-beziehung.

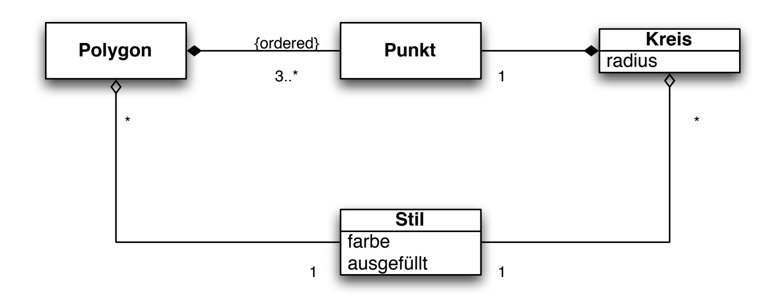


Abbildung 22: Aggregationsstruktur

zwischen aggregation und komposition wird aber erst beim implementieren bedeutsam: ein polygonzugobjekt bzw. ein kreisobjekt enthält die beteiligten punkte *direkt*, während es nur eine *referenz* auf den darstellungsstil enthält. In der modellierungsphase haben solche fragen im grunde nichts zu suchen.

4.2.3 Vererbung

Eine klasse kann spezialfall einer anderen klasse sein und somit von dieser klasse eigenschaften (attribute und methoden) erben. Wir sprechen dann von *subklasse* und *superklasse* und von einer *is-a*-relation. In abb. 23 sind der firmenkunde (`Corporate Customer`) und der privatkunde (`Personal Customer`) spezialfälle des allgemeinen kunden (`Customer`), mit dem sie sich die attribute „name“ und „adresse“ und die methode „kreditrahmen“ teilen. Darüber hinaus haben sie jedoch je eigene attribute und methoden.

Die abbildung zeigt auch noch einen weiteren aspekt von assoziationsbeziehungen: die *navigierbarkeit*, beispielsweise an der assoziation zwischen `Order` und `Customer`. Wenn wir diese navigierbarkeit in begriffen der analyse betrachten, dann bedeutet der pfeil, dass ein `Order`-objekt stets „weiss“, zu welchem `Customer`-objekt diese bestellung gehört. Umgekehrt braucht das `Customer`-objekt nicht zu „wissen“, welche bestellungen damit verbunden sind. In begriffen der implementierung heisst die navigierbarkeit, dass ein `Order`-objekt eine referenz auf ein `Customer`-objekt haben wird, aber nicht umgekehrt. Diese sichtweise findet sich auch in der annotation „`Order.customer.creditRating`“ wieder. In der klasse `Order` gibt es gar kein feld `customer`, aber wegen der navigierbarkeit darf man annehmen, dass es eine entsprechende referenz geben wird.

Wir reden hier von einer *vererbungsbeziehung* („inheritance relation“); auch der begriff *klassifikationsbeziehung* ist üblich. Da eine subklasse eine *spezialisierung* der superklasse ist bzw. die superklasse eine *generalisierung* der Subklas-

se, spricht man auch von einer *GenSpec-beziehung*. Die durch vererbungs- bzw. klassifikationsbeziehungen induzierte struktur zwischen den klassen nennen wir die *klassifikationsstruktur*.

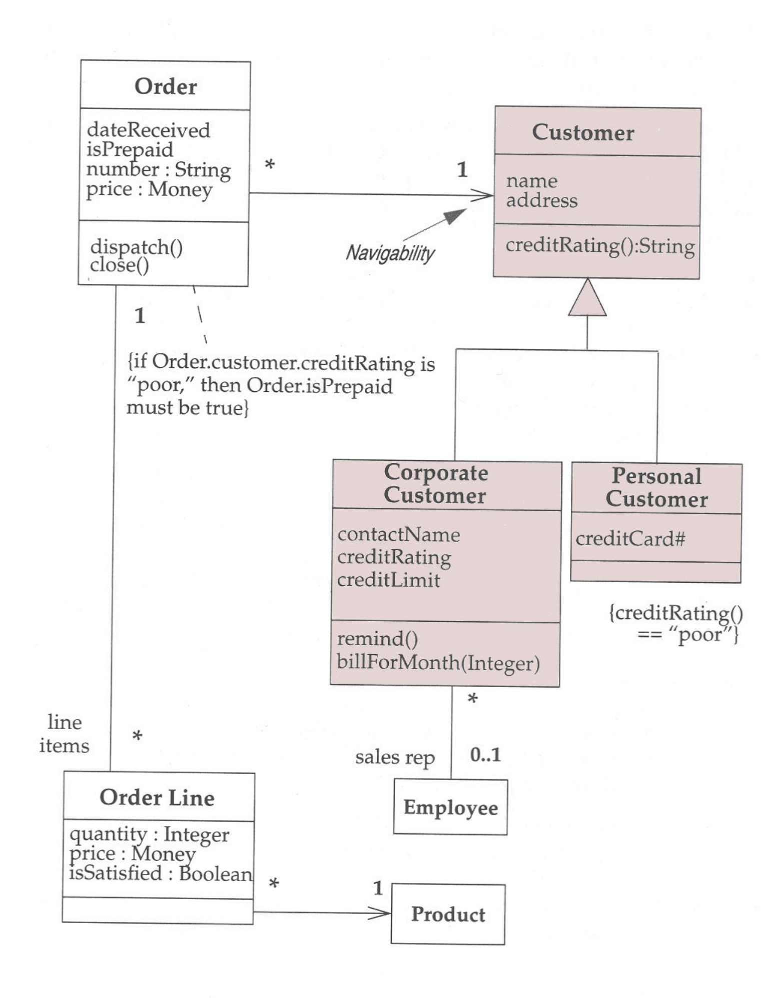


Abbildung 23: Vererbungsstruktur

Gamma u. a. (1995) weisen zu recht darauf hin, dass es in der praxis häufig vorkommt, dass man eine entwurfsentscheidung treffen muss, ob beziehungen zwischen komplexen klassen in form von aggregationen bzw. kompositionen oder von GenSpec-beziehungen modelliert werden sollen. Vom standpunkt des geheimnisprinzips her ist die aggregation, wo immer anwendbar, zu bevorzugen: Hier wird ein komplexes objekt dargestellt als zusammengesetzt aus mehreren anderen objekten, wobei deren schnittstellen und das mit ihnen verbundene geheimnis erhalten bleiben. Es ist also niemals notwendig, „in die objekte hineinzuschauen"; die implementierung jedes teilobjektes kann

jederzeit geändert werden, ohne dass sich am zusammengesetzten objekt etwas ändern muss. Vererbung bricht im gegensatz dazu das geheimnisprinzip stellenweise auf: Jede subklasse „sieht" naturgemäss den kompletten code ihrer oberklasse und kann auf jedes attribut und jede auch interne methode direkt zugreifen. Es kann dabei sehr wohl geschehen, dass änderungen in der implementierung der oberklasse zwangläufig änderungen in den unterklassen nach sich ziehen.

Vererbung ist also, obwohl in gewissem masse das herausstechendste merkmal der OO-techniken ein eher problematisches konzept, das nur nach sorgfältiger abwägung eingesetzt werden sollte. Zu beginn der modellierung ist es von geringem wert und sollte da noch nicht eingesetzt werden.

Vererbung wird im wesentlichen mit zwei zielen eingesetzt:

1. Zur formalisierung einer can-be-used-as-beziehung.

2. Zur wiederverwertung von implementationsteilen.

Das erste ziel bezieht sich deutlich auf die rollen, die objekte spielen, bzw. in der sie von algorithmen behandelt werden. Diese art der generalisierung kann erst nach einer verhältnismässig weit fortgeschrittenen *dynamischen* modellierung (s. abschn. 4.3) vernünftig durchgeführt werden. Das zweite ziel gehört bereits zur implementationstechnik und soll auf keinen fall in den entwurf zurückwirken.

Vererbungsbeziehungen erzeugen sehr starke abhängigkeiten zwischen systemteilen und müssen deshalb so spät als möglich eingeführt werden.

Durch das vererbungskonzept bei objektorientierten sprachen kommt automatisch eine gewisse form der *polymorphie*[14] ins spiel (Cardelli und Wegner 1985), nämlich die sog. *inklusionspolymorphie*: An jeder stelle, wo wir ein objekt einer bestimmten klasse C als argument einer methode erwarten, kann auch jedes objekt jeder unterklasse von C übergeben werden. Da es unterklassen erlaubt ist, statt des kommentarlosen erbens von methoden aus einer oberklasse diese auch zu überschreiben, stellt sich die interessante frage nach der *bindung*, d. h. der auswahl einer bestimmten variante der methode aus mehreren möglichkeiten innerhalb der vererbungshierarchie. Besonders einfach und übersichtlich ist die sog. *statische bindung* (z. b. bei Oberon-2). Hier ist bereits der compiler in der lage, diese auswahl zu treffen. Komplizierter wird es etwa im fall von Smalltalk, wo erst zur laufzeit feststehen kann, auf welcher stufe der vererbungshierarchie eine methode tatsächlich ausgeführt wird. Hier sprechen wir von *dynamischer bindung*.

[14] Vgl. meine bemerkung über die vermischung von klassen und typen!

Bei der *einfachen vererbung* hat keine klasse mehr als eine oberklasse. Der nutzen der *mehrfachvererbung*, wo eine klasse von mehreren oberklassen erben kann, ist theoretisch und praktisch umstritten; einige OO-programmiersprachen verbieten die mehrfachvererbung. Java bietet mit seinen `interfaces` eine pragmatische lösung an: eine klasse kann mehrere verschiedene `interfaces` implementieren.

4.3 Dynamische modellierung

Wir haben im vergangenen abschnitt das zu entwickelnde system ledig *statisch* modelliert; genau so wichtig ist die *dynamische modellierung*. Hier geht es um die beschreibung von *vorgängen* im system, die einen gewissen *zeitverlauf* haben und um *zustandsveränderungen* im system. Dazu gehört auch die darstellung der sogenannten „objektgeschichte“ (*object life cycle*), d. h. die beschreibung der stellen und zeitpunkte, wo objekte erzeugt und vernichtet werden.

Etwas völlig anderes wäre eine zusicherung, dass gewisse attribute eines objektes über seine lebensdauer hinweg konstant bleiben (z. b. die in einen personalausweis eingedruckten personenmerkmale). Auch dies ist zwar im grunde eine dynamische eigenschaft, wird aber besser durch annotationen in einem klassendiagramm verankert werden.

Wichtige beschreibungsmittel in der UML zur dynamischen modellierung sind:

- Zustandsübergangsdiagramme (transitionsdiagramme, im prinzip: endliche automaten, s. abb. 24)
- Sequenzdiagramme, s. abb. 25

Die zustandsübergangsdiagramme entsprechen dem konzept des endlichen automaten, das aus der theoretischen informatik bekannt ist, genauer gesagt: der *generalised sequential machine*. Es gibt hier eine endliche menge von *zuständen* und dazwischen *zustandsübergänge* (transitionen). Einer der zustände ist (durch einen schwarzen ausgefüllten kreis) als *anfangszustand* ausgezeichnet (im beispiel: `Checking`); zustände, aus denen keine transitionen wegführen, gelten als *endzustände* (im beispiel: `Cancelled` und `Delivered`). Jeder zustand hat einen namen und möglicherweise auch noch (unter dem strich) *aktionen*, die ausgeführt werden, wenn das system in diesem zustand ist. Manche transitionen werden durch bestimmte *ereignisse* ausgelöst (im beispiel: `Item Received`), andere können sich *spontan* ereignen (im beispiel die transition von `Checking` nach `Dispatching`). Darüber hinaus können transitionen aber auch von einem *guard* in eckigen klammern bewacht werden, also einer bedingung,

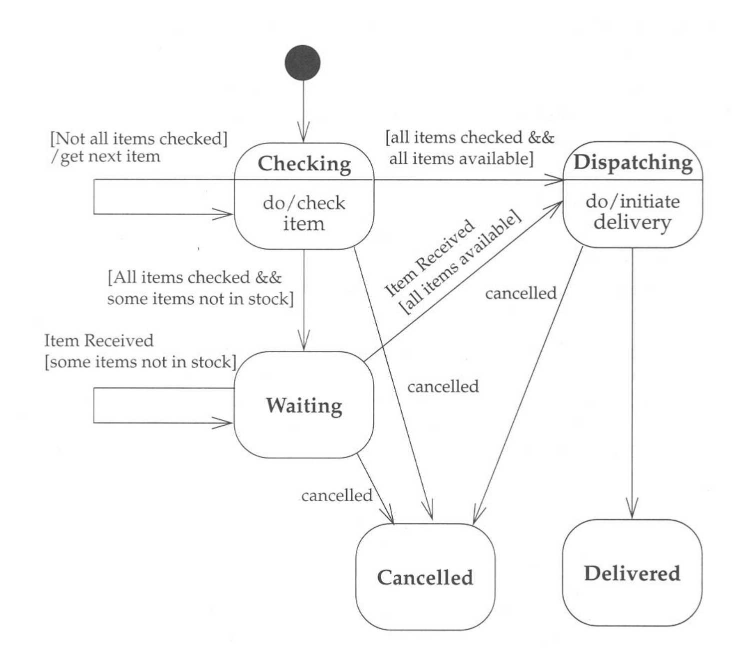


Abbildung 24: Transitionsdiagramm

die erfüllt sein muss, bevor die transition stattfinden kann (im beispiel: `[all items available]`). Des weiteren kann auch mit einer transition eine aktion verbunden sein (im beispiel: `get next item`), die während des zustandsübergangs ausgeführt wird, also jedenfalls, bevor der folgende zustand wirksam wird.

Beim sequenzdiagramm (abb. 25) ist zu beachten, dass hier *objekte*, keine klassen, in den kästchen vorkommen. Die senkrechte achse ist die zeit-achse, in der horizontalen werden nachrichten notiert, die sich objekte schicken. Sequenzdiagramme sind von den *message sequence charts (MSCs)* der ITU-T abgeleitet; sie zeigen, welche abläufe von interaktionen zwischen objekten in einem system *möglich* sind, sagen aber nicht aus, dass diese abläufe stets in dieser form bis zu ende durchgeführt werden müssen. Auch hier können interaktionen durch *guards* bewacht werden.

4.4 Model driven architecture

Nachdem nun die wesentlichen ingredienzien der UML vorgestellt sind, kann die frage gestellt werden, wie diese beschreibungsmittel denn zur objektorientierten modellierung politisch korrekt eingesetzt werden sollen. Vor dem hintergrund, eine vernünftige *architektur* gewinnen zu können, hat die OMG

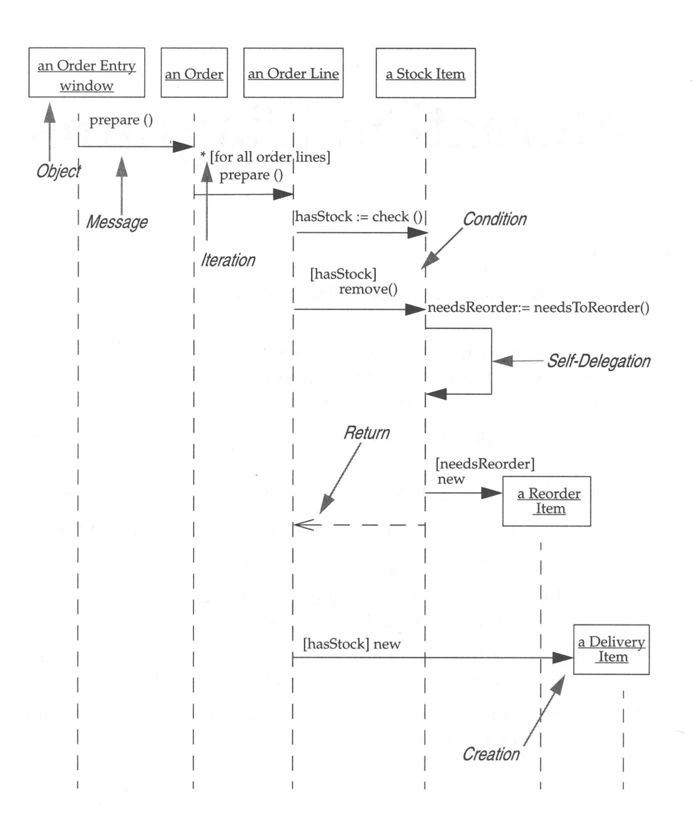


Abbildung 25: Sequenzdiagramm

die *Model Driven Architecture (MDA)* vorgeschlagen. Der gedanke dabei ist es, ein systemmodell in vier getrennten schichten darzustellen (Kleppe u. a. 2003):

Computation Independent Model (CIM) Hier werden die geschäftsprozesse auf hohem abstraktionsniveau beschrieben, ohne dabei rücksicht darauf zu nehmen, an welchen stellen hier computer bzw. programme zum einsatz kommen.

Platform Independent Model (PIM) In diesem modell werden die geschäftsmodell-anteile, die von computern übernommen werden, gesondert dargestellt, ohne dabei jedoch auf details einzugehen, welche kombination von hardware und software hier verwendet werden soll.

Diese beiden schichten sind rein *problemorientiert*, sie nehmen keine details irgendeiner lösung vorweg.

Platform Specific Model (PSM) Von dieser schicht an geht es nicht mehr so sehr um das problem als vielmehr um seine lösung. Es liegt deshalb ein kreativer schritt im übergang vom PIM zum PSM („low-level design“), hier werden eine menge entwurfsentscheidungen getrofffen.

Code model Dieses modell entspricht bereits der implementierung im ausprogrammierten code.

Customer
+title: String +name: String +dateOfBirth: Date

Abbildung 26: Kundenklasse im PIM

Als Beispiel zeigt abb. 26 ein PIM einer kundenklasse `Customer` und abb. 27 das entsprechende auf Java abgezielte PSM (Kleppe u. a. 2003). Diese abbildungen zeigen auch die in abb. 20 noch nicht genutzte möglichkeit, datentypen und sichtbarkeiten (`+` für öffentlich sichtbar (`public`) und `-` für privat (`private`)) zu spezifizieren. Im vergleich der beiden abbildungen zeigt sich ein typisches phänomen, das häufig in der system-modellierung angetroffen wird: In der analysephase, zu der das PIM gehört – und ebenso in der nachfolgenden grobentwurfsphase – ist es durchaus üblich, die attribute einer klasse als öffentlich zu betrachten und sich an allen möglichen stellen darauf zu beziehen. In einem PSM stellt sich dies als schlechte idee heraus: Damit eine klasse

Customer
-title: String -name: String -dateOfBirth: Date
+getTitle(): String +setTitle(title: String) +getName(): String +setName(name: String) +getDateOfBirth(): Date +setDateOfBirth(d: Date)

Abbildung 27: Kundenklasse in einem Java-PSM

ihre invarianten garantieren kann, sollten die attribute unbedingt privat sein. Dann braucht man methoden, die über die attribute auskunft geben („`get`-methoden“) und die attribute *unter der kontrolle der klasse* verändern können („`set`-methoden“). Beachte, wie sich ausserdem in der notation der methoden die angleichung an die programmiersprache niedergeschlagen hat.

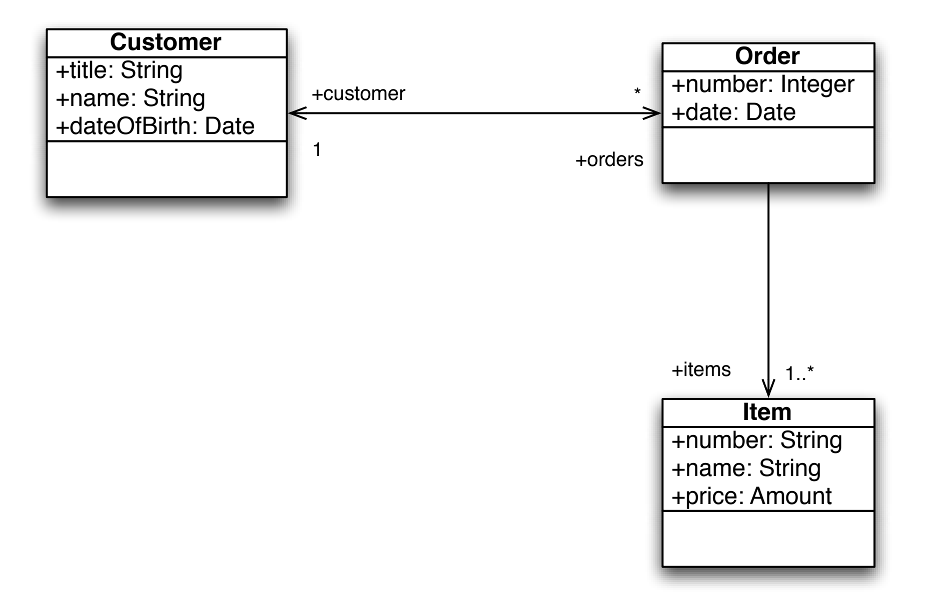


Abbildung 28: Bestellung im PIM

Der unterschied zwischen PIM und PSM wird noch deutlicher, wenn man ein komplizierteres klassendiagramm wie in abb. 28 betrachtet. Hier wurde die kundenklasse noch um eine bestellungsklasse (`Order`) mit einzelnen artikeln (Klasse `Item`) und die dazwischen bestehenden beziehungen mit ihren

Customer
-title: String
-name: String
-dateOfBirth: Date
-orders: Set
+getTitle(): String
+setTitle(title:String)
+getName(): String
+setName(name: String)
+getDateOfBirth: Date
+setDateOfBirth(d:Date)
+getOrders(): Set
+setOrders(o: Set)

Order
-number: Integer
-date: Date
-customer: Customer
-items: Set
+getNumber(): Integer
+setNumber(n: Integer)
+getDate(): Date
+setDate(d: Date)
+getCustomer(): Customer
+setCustomer(c: Customer
+getItems(): Set
+setItems(s: Set)

Item
-number: String
-name: String
-price: Amount
+getNumber(): String
+setNumber(n: String)
+getName(): String
+setName(s: String)
+getPrice(): Amount
+setPrice(p: Amount)

Abbildung 29: Bestellung im Java-PSM

kardinalitäten und rollen erweitert. Im Java-PSM (abb. 29) sind die beziehungen zwischen den klassen erschreckenderweise gar nicht so deutlich sichtbar: Erst auf den zweiten blick sieht man in der `Customer`-klasse ein attribut `orders` und in der `Order`-klasse umgekehrt ein attribut `customer`. In instanzen dieser klassen wird man an diesen stellen objektreferenzen vorfinden. Entsprechend findet sich in der `Order`-klasse ein attribut `items`, das auf die artikel einer bestellung verweisen kann. Ein umgekehrter verweis von einem artikel auf eine bestellung ist nicht vorhanden; in der tat ist die beziehung zwischen `Order` und `Item` im PIM ja auch nur in einer richtung als navigierbar gekennzeichnet.

Die OMG redet bei der MDA sehr gerne von *transformationen* zwischen diesen klassen von modellen, die dann womöglich auch noch (halb-) automatisch durch werkzeuge durchgeführt werden. Klar ist jedoch, dass der übergang vom CIM zum PIM nicht automatisch sein kann, weil das CIM viel zu wenig formal ist – es besteht in der regel aus sehr viel umgangssprache – und weil in der festlegung, welche teile des gesamtsystems computerisiert werden sollen, eine wichtige und schwer zu treffende entscheidung liegt. Auch der übergang vom PIM zum PSM kann sicher nur in bescheidenem masse durch werkzeuge unterstützt werden, auch wenn Kleppe u. a. (2003) hier grosse erwartungen wecken. Eine rücktransformation von einem PSM zu einem PIM ist offensichtlich niemals möglich, wie schon das oben angeführte beispiel klarmacht. Ebenso klar ist aber andererseits, dass der übergang vom PSM zum code und zu beträchtlichem anteil auch umgekehrt sehr gut durch werkzeuge unterstützt werden kann.

Im universitären *FUJABA*-projekt („From UML to Java and back“, `http://www.fujaba.de`) wurden werkzeuge entwickelt, die ein solches *round trip engineering* mit einem mehrfachen wechsel von modellierungsebenen in beiden richtungen ermöglichen.

4.5 Prozess einer objektorientierten modellierung

Ein gängiges paradigma der objektorientierten systemerstellung ist es, aus einer beschreibung des systems als bestandteil der realen welt die hier involvierten objekte, ihre attribute und methoden sowie deren vielfältige beziehungen zueinander herauszuschälen und zu modellieren.

Zunächst wird ein system von aussen betrachtet und die benutzung des systems durch den anwender beschrieben. Dabei wird das system als objekt aufgefasst und die interaktion der benutzer mit dem system als austausch von nachrichten.

Eine besondere rolle spielen in diesem zusammenhang die sog. anwendungsfälle (*use cases*). Abgesehen von der (relativ wenig aussagekräftigen) möglichkeit, solche anwendungsfälle in der UML graphisch zu veranschaulichen, handelt es sich hierbei um kleine skripte („drehbücher“, „*stories*“, „*scenarios*“), welche in möglichst standardisierter form („formular“) handlungen beschreiben, welche benutzer mit dem system durchführen wollen. Dabei wird natürliche sprache verwendet, die aber meist zahlreiche möglichkeiten zu missverständnissen bietet. Berry und Kamsties (2005) weisen darauf hin, dass viele dieser missverständnisse vermieden werden können, indem man eine grössere sprachliche sorgfalt walten lässt[15].

Während des objektorientierten systementwurfs nehmen wir dann einen *paradigmenwechsel* vor: jetzt wird das zu gestaltende system von innen her betrachtet. Die darstellung wird erweitert um weitere klassen, die für die implementierung hilfreich oder gar notwendig sind. Hier erhält der begriff der systemarchitektur grösste bedeutung. Systemarchitektur in OO-systemen besteht aus der klassenstruktur, die ausgehend vom modell der analyse um zusätzliche elemente erweitert wird. Zu den klassen des anwendungsbereichs kommen technische klassen hinzu, die das in der anforderungsspezifikation definierte verhalten des systems implementieren. auf diese weise entstehen immer weitere klassen, bis wir schliesslich bei einem ausführbaren system angelangt sind.

Da die klassen zunächst aus der analysephase stammen, sehen praktisch alle entwurfsmethoden massnahmen zur verbesserung der klassenhierarchie vor, wie etwa die einführung von *abstrakten klassen* an geeigneten stellen, die überprüfung von wohlgeformtheitsbedingungen oder eine robustheitsanalyse. Auch die wiederverwendung von klassen und entwurfsmustern wird an dieser stelle häufig einbezogen.

[15] Als beispiel dient der satz: „Alle lampen in den räumen haben einen einzigen schalter“. Was soll das heissen? Gibt es einen einzigen schalter, der alle lampen in allen räumen bedient, gibt es für jeden raum einen einzigen schalter, der alle lampen in diesem raum bedient oder gibt es für jede lampe in jedem raum genau einen schalter?

Im idealfall enthält der objektorientierte entwurf alle details, die für die erstellung eines programms notwendig sind. Wenn dann auch die programmiersprache alle konzepte der verwendeten notationen unterstützt, so reduziert sich die übersetzung vom entwurf zum code fast auf eine 1:1-abbildung. Viele objektorientierte CASE tools verfügen daher über einen codegenerator für C++ oder Smalltalk. Natürlich sind in der praxis häufig optimierungen erforderlich oder komplexe algorithmen zu implementieren, die sich mit einer graphischen notation nicht ausdrücken lassen. Von einem guten CASE tool darf man erwarten, dass manuelle änderungen am erzeugten code automatisch in das klassenmodell zurückübersetzt werden können.

Genauere handlungsanleitungen liefert das buch von Coad und Yourdon (1991). Typisches vorgehen ist es dabei, zunächst eine *sichtung* durchzuführen, in der potentielle kandidaten vor dem hintergrund ihrer aufnahme in das modell gesammelt werden. Die liste dieser kandidaten wird dann einer kritischen *bewertung* unterzogen, wobei unter umständen elemente ausgesondert werden.

In diskussionen über objektorientierte programmierung wird häufig ungenau formuliert: es wird von „objekten" geredet, wenn in wirklichkeit klassen gemeint sind und umgekehrt[16]. Grund für diese verwirrung ist, dass hinter einem objekt, das wir auffinden, immer die klasse aller objekte steht, die diesem ähnlich sind, und hinter einer klasse immer die menge aller objekte, die man als exemplare (instanzen) davon generieren kann.

4.5.1 Definition der klassen

Der erste schritt bei der analyse besteht darin, die relevanten klassen des systembereichs aufzufinden und zu beschreiben. Hierzu werden zunächst informelle dokumente zur problembeschreibung (vor allem die „anwendungsfälle") sorgfältig gelesen und ausgewertet und die hierin enthaltenen angaben im gespräch mit experten vervollständigt.

Kunden und informatiker tendieren im allgemeinen dazu, lieber vorgänge (prozesse) als situationen zu beschreiben. Die kunst der statischen modellierung besteht genau darin, aus beschreibungen von abläufen einen systemzustandsbegriff, ein datenmodell, herauszuschälen. Die *verb-substantiv-methode* bietet hier einen geeigneten ansatzpunkt: Substantive in problembeschreibungen deuten auf eher auf klassen und ihre attribute, verben eher auf methoden und assoziationen hin. Allerdings kann diese methode auch in die irre führen: Nicht jedes substantiv eignet sich als klasse und nicht jedes substantiv ist

[16]Auch dieses skriptum ist nicht frei davon! Hinweise werden stets dankend entgegengenommen!

für den zu modellierenden ausschnitt des systems wirklich relevant. Hier ist sorgfältige abwägung gefragt.

Ausserdem sollte man der suggestion der klassennamen nicht erliegen und „ontologisch“ modellieren: Wir reden im kontext des entwurfs von software in der regel nur über konstrukte innerhalb des computers, d. h. *nicht* über die reale welt. Ein objekt a der klasse *auto* ist nicht mein auto, sondern es *beschreibt* (in diesem fall) mein auto.

Klassen, welche namen physikalischer gegenstände tragen, stehen im kern meist nur für datensätze, die diese gegenstände beschreiben[17]. Modelliert werden nur die beziehungen zwischen datensätzen. Es lohnt, sich diesen sachverhalt zuweilen vor augen zu führen, gerade, weil es soviel schlechte OO-literatur gibt, die diesen unterschied nicht macht.

Aufgefundene klassen werden mit ihren attributen und methoden textlich und graphisch erfasst. Auf der suche nach relevanten klassen konzentriert man sich am besten auf

- Strukturen, organisatorische einheiten
- Rollen, die personen übernehmen können
- Andere systeme, mit denen das zu entwickelnde system zusammenarbeitet
- Externe geräte
- Ereignisse, die gespeichert werden müssen
- Orte

Als namenskonvention sollte unbedingt eingehalten werden, dass klassennamen im *singular* formuliert werden! Also nicht „Studenten“, sondern „Student“, nicht „Vorlesungen“, sondern „Vorlesung“. Selbstverständlich steht die klasse „Student“ in gewissem sinne für *alle* studenten, aber wie schon gesagt, ist eine klasse eine art schablone oder stempel, um *exemplare* bzw. *instanzierungen* zu erzeugen, und diese erhält man jeweils einzeln, z. b. durch eine formulierung wie `new Student`. In einer solchen formulierung würde der plural merkwürdig aussehen. Berry und Kamsties (2005) weisen ausserdem darauf hin, dass der plural in spezifikationen anlass zu missverständnissen bieten kann[18]. Übrigens ist es auch konvention, die klassennamen mit einem

[17] Ein berühmtes bild von Magritte zeigt eine fotorealistisch gemalte pfeife mit der unterschrift: „Ceci n’est pas une pipe.“ In der tat, es ist wirklich keine pfeife, es ist das abbild einer pfeife!

[18] Vergleiche die beiden sätze: „Studenten belegen 6 vorlesungen“ und „Studenten belegen hunderte von vorlesungen“. Nur das „weltwissen“ macht deutlich, dass in diesem fall im ersten satz jeder einzelne student gemeint ist und im zweiten die gesamtheit aller studenten. „Jeder student belegt 6 vorlesungen“ ist die deutlich bessere formulierung für den ersten satz.

grossbuchstaben beginnen zu lassen. Von Smalltalk stammt die weitere konvention, einzelne objekte einer klasse durch den klassennamen mit einem vorangestellten „a/an“ zu bezeichnen, also `aStudent`, `anExercise` etc.

Die solcherart aufgefundenen kandidaten werden anhand der folgenden kriterienliste bewertet:

- Muss das system über objekte dieser klasse informationen speichern oder methoden für sie bereitstellen? Falls keiner dieser beiden fälle vorliegt, ist die klasse nicht relevant und kann wegfallen. Falls ja, welche informationen (attribute der klasse) und methoden sind dies?
- Hat die klasse mehr als nur ein einziges attribut? Falls nein, handelt es sich möglicherweise nicht um eine selbständige klasse, sondern bloss um ein attribut einer anderen klasse.
- Lassen sich die attribute der klasse aus denen anderer klassen berechnen? Falls ja, sollte diese klasse in der analysephase nicht berücksichtigt werden, da sie in der entwurfsphase einfach hinzugefügt werden kann.
- Entwurfsspezifische klassen (menüs, fenster, laufwerke etc.) sollten getrennt notiert bzw. markiert werden und erst in der entwurfsphase berücksichtigt werden.

Zur bezeichnung einer klasse wird als name grundsätzlich ein *substantiv* oder aber eine kombination von adjektiv und substantiv verwendet.

4.5.2 Einführung der struktur

Zu diesem zeitpunkt konzentrieren wir uns auf die beiden wichtigsten strukturierungsprinzipien der OOA, nämlich assoziationsstruktur und klassifizierungsstruktur. Sinnvoll ist es, zuerst nach *assoziationsstrukturen* zu suchen, da diese weniger festlegungen für das softwaresystem im ganzen implizieren, und dann erst klassifikationsstrukturen. Diese letzteren findet man durch die beantwortung der folgenden fragen für jede klasse:

- Welche spezialisierungen der klasse kommen innerhalb des problemfelds vor? Haben die spezialisierungen besondere attribute oder methoden? Spiegelt die spezialisierung einen sachverhalt der realen welt wieder? Auf der suche nach spezialisierungen hilft die folgende frage weiter: Ist die aufgefundene menge von attributen/methoden für jedes vorkommen der klasse relevant? Falls einige attribute/methoden nicht in allen verwendungen der klasse sinnvoll sind, kann es richtig sein, diejenigen, die überall sinnvoll sind, in einer oberklasse zusammenzufassen.

- Welche verallgemeinerungen (generalisierungen) einer klasse kommen innerhalb des problemfelds vor? Ist die betrachtete klasse spezialfall einer anderen? Auf der suche nach generalisierungen hilft die folgende frage weiter: haben klassen gemeinsame attribute/methoden? Dann kann es richtig sein, diese in einer oberklasse zusammenzufassen.

Besonders mit der frühzeitigen einführung einer vererbungsstruktur kann man sich viel ärger einhandeln! Zwar werden die OO-ansätze allgemein wegen ihrer angeblichen flexibilität gerühmt, es hat sich jedoch herausgestellt, dass nichts so starr ist wie eine einmal eingeführte klassenhierarchie. Hieran lässt sich nur noch sehr schwer später rütteln. Es ist deshalb weise, zunächst einmal auf klassifikationen keinerlei rücksicht zu nehmen; wenn sie später eingeführt werden, sind die folgenden punkte unbedingt zu beachten:

- Die bildung von oberklassen hat nur dann eine berechtigung, wenn es situationen gibt, in denen objekte mehrerer unterklassen gleichartig behandelt werden.
- Jede unterklasse muss den kontrakt (*„design by contract"!*) der oberklasse einhalten. Insbesondere muss sie alle attribute und methoden der oberklasse besitzen.

In der nachfolgenden prüfung und bewertung werden aus den möglichen generalisierungen und spezialisierungen die nicht als relevant betrachteten wiederum ausgeschieden. Durch analoge fragen findet man auch aggregationsstrukturen heraus.

4.5.3 Aufteilung des modells in subsysteme

Unabhängig von der strukturierung des problemfelds durch klassifikation anhand gemeinsamer attribute und methoden kann die schrittweise erfassung der beschreibung des problemfelds für den leser durch eine einteilung in subsysteme (sachgebiete, „subjects" bei Coad/Yourdon) erleichtert werden. Psychologische forschungen haben ergeben, dass das menschliche kurzzeitgedächtnis maximal 5–7 dinge gleichzeitig erfassen kann. Dementsprechend versucht man, subsysteme so zu definieren, dass höchstens 7 klassen oder subsysteme in einem subsystem zusammengefasst werden.

4.5.4 Definition der attribute

Bereits im ersten schritt hatten wir klassen zusammen mit ihren attributen erfasst. Hier geht es nun darum, diese attribute zu konsolidieren und vor allem

bezüglich ihres datentyps festzulegen. Dabei muss auch bedacht werden, dass nicht alle möglichen attribute einer betrachteten klasse tatsächlich relevant sind. Innerhalb der klassifikationsstruktur sind attribute so hoch wie möglich anzusiedeln.

4.5.5 Definition der methoden

Auch methoden werden in der klassifikationsstruktur so hoch wie möglich angesiedelt.

Tendenziell sollten methoden erst nach der statischen modellierung gesucht und charakterisiert werden. Einige methoden drängen sich aber bereits während der statischen modellierung geradezu auf, und können bereits dann festgehalten und charakterisiert werden. Da die meisten OO-prozesse iterativ sind, ist es sowieso schwierig, statische und dynamische entwurfstätigkeiten zeitlich voneinander zu trennen. Trotzdem bleibt auch hier die faustregel „die daten zuerst!" gültig.

Häufig lässt sich die leistung einzelner methoden einfach in natürlicher sprache charakterisieren, indem man vor- und nachbedingungen und veränderte zustände beschreibt. Natürlichsprachige oder informell mathematische beschreibungen können wie einschränkungen oder andere dynamische aspekte als notizen oder fussnoten notiert werden.

4.6 Muster

In etablierten ingenieurwissenschaften wie maschinenbau, elektrotechnik, bauingenieurwesen oder architektur werden erfahrungswerte aus handbüchern („kochbüchern") verwendet. Diese erfahrungen basieren auf immer wieder erfolgreich erprobten lösungen. Eine wiederverwendung solcher lösungen spart nicht nur gedankenarbeit bei der konstruktion, sondern erleichtert auch in erheblichem mass das verständnis von programmen, wenn diese gewartet werden müssen. Nicht umsonst sagt Alan Perlis (1982) schon an zehnter stelle seiner „Epigrams on Programming":

> *Komm schnell in eine spur: Erledige die gleichen prozesse auf die gleiche Weise. Sammle redewendungen. Normiere. Der einzige unterschied (!) zwischen dir und Shakespeare ist die grösse der liste seiner redewendungen — nicht die grösse seines vokabulars.*

Bei der konstruktion von maschinen wird auf konstruktionskataloge zurückgegriffen, in denen muster für standardteile gesammelt sind. In der elektrotechnik werden bauteile anhand bekannter muster zu grösseren einheiten

(baugruppen) kombiniert. Zusätzlich beschreiben sammlungen von abschätzungen das verhalten der baugruppen. Durch diese abschätzungen lassen sich mit weitaus geringerem aufwand lösungen finden.

Ein fernziel der softwaretechnik ist die entwicklung von handbüchern als sammlungen wiederverwendbarer erfahrungen. Objektorientierte muster sind ein schritt in diese richtung. Sie ermöglichen zum einen, dass erfolgreiche architekturentwürfe wiederverwendet werden können. Zum anderen erleichtern sie die dokumentation von software, indem sie zur beschreibung der architektur verwendet werden können.

Das konzept, bei der entwicklung von systemen auf wiederverwendbare muster zurückzugreifen, ist zuerst von C. Alexander u. a. (1977) formuliert worden (s. a. anh. E, p. 240):

> *Jedes muster beschreibt ein problem, das immer und immer wieder in unserer umgebung auftritt, und es beschreibt den kern der lösung zu diesem problem in einer weise, dass man die lösung millionen von malen benutzen kann, ohne jemals die gleiche sache zweimal tun zu müssen.*

Alexander bezieht sich dabei auf die entwicklung von gebäuden und städten, aber seine aussage kann als grundlegende definition von mustern in der objektorientierten software-entwicklung verwendet werden. Alexander fordert, dass jeder architekt sich eine eigene „sprache" solcher muster zulegen muss, in der er sich ausdrückt, um einerseits seinen persönlichen, wiedererkennbaren stil zu finden und andererseits nicht immer wieder von neuem das rad erfinden zu müssen.

Objektorientierte muster sind lösungsvorlagen für problemstellungen, die immer wiederkehren. Dabei beschreibt ein muster nur den abstrakten lösungsansatz für den entwurf. Von ihm können immer wieder neue, einer konkreten problemstellung angepasste lösungsvarianten abgeleitet werden.

Das erste allgemein verwendbare objektorientierte muster war das von Krasner und Pope (1988) im zusammenhang mit der entwicklung von Smalltalk veröffentlichte „model-view-controller"-muster; es beschreibt einen entwurf zur entwicklung von graphisch interaktiven systemen. Es geht zurück auf ein von Trygve Reenskaug (1979) ein paar Jahre früher beschriebenes *thing-model-view-editor*-muster.

Hier werden die klassen einer graphischen benutzeroberfläche in drei gruppen eingeteilt: *model*, *view* und *controller*.

- Das *modell* beinhaltet die zentralen zustandsdaten des modellierten systems und alle methoden, die zur beschreibung der zustandsänderungen des modells notwendig sind, jedoch keinerlei methoden für die evtl. interaktion mit dem benutzer.

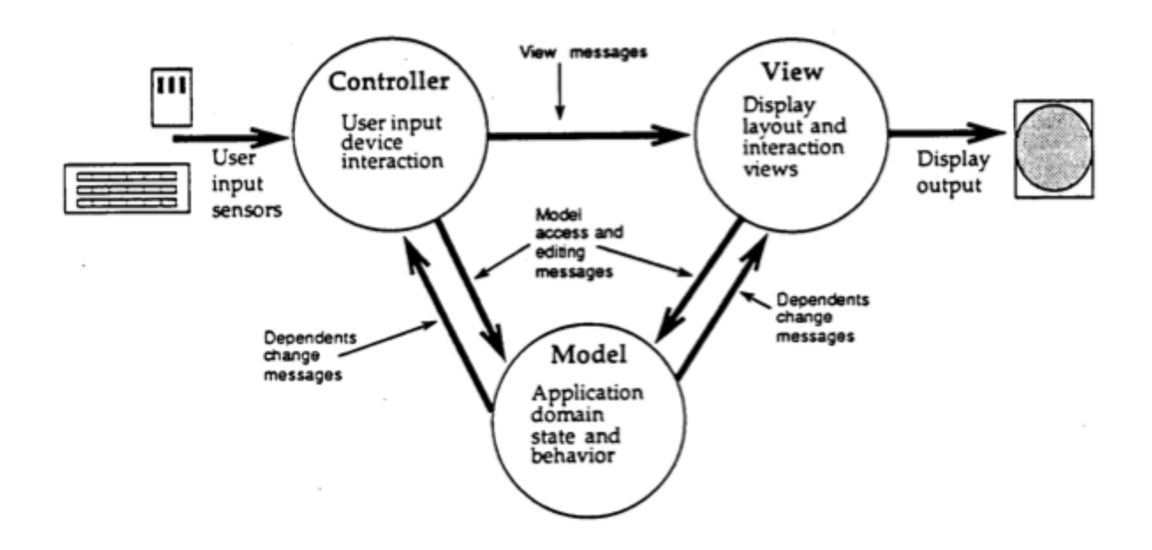


Abbildung 30: Model-View-Controller (MVC)

- Die *ansicht* (*view*) zeigt dem benutzer einen ausschnitt der informationen auf dem bildschirm an. Es ist durchaus normal, dass ein system mehrere verschiedene ansichten zu einem modell enthält. Die für die erzeugung der ansicht notwendigen daten bekommt die ansicht vom modell. Andererseits registrieren sich alle ansichten beim modell, so dass dieses, wenn sich etwas ändert, die verschiedenen ansichten benachrichtigen kann. Elemente der benutzer-interaktion (tastatureingaben, mausklicks) werden nach heutiger sicht des musters und im gegensatz zu Krasner und Pope von der ansicht entgegengenommen und an die steuerung weitergeleitet, die dann entscheidet, was nun zu geschehen hat.

- Die steuerung (*controller*) interpretiert die grafischen eingaben des benutzers in der ansicht, passt die ansicht dementsprechend an und sendet möglicherweise änderungsaufforderungen an das modell.

Insgesamt bleibt Reenskaug bei der erklärung des musters auf einer sehr allgemeinen ebene; Krasner und Pope beschreiben etwas genauer die vorgänge. In der „bibel" der objektorientierten muster von Gamma u. a. (1995) kommt das eher komplizierte MVC nur als historische referenz vor; wesentliche aspekte davon lassen sich griffiger durch das beobachter-muster, (*observer pattern*, 4.7) beschreiben.

4.6.1 Dokumentation objektorientierter muster

Damit muster allgemein verständlich in handbüchern gesammelt werden können, müssen sie einheitlich dokumentiert werden. Es gibt also auch ein „muster zur beschreibung von mustern“, wie in der „pattern-bibel“ von Gamma u. a. (1995) vorgeführt wird. Dieses buch, obwohl schon zehn jahre alt, ist immer noch eine lesenswerte quelle für jeden, der sich für objektorientierte muster interessiert. Zwei probleme mit diesem buch sollen aber kritisch angemerkt werden:

- Die grafischen darstellungen sind nicht in UML, sondern in einer abweichenden (und in bezug auf *abstrakte klassen* typographisch heiklen) älteren notation.
- Zur entstehungszeit des buchs gab es keine programmiersprache, die *interfaces* als eigenständiges konzept besass; deshalb wird vieles mit hilfe von *abstrakten klassen* dargestellt, was sich heute viel besser durch *interfaces* erklären liesse.

Das muster zur beschreibung von mustern bei Gamma u. a. (1995) ist folgendes:

- **Name und klassifikation des musters**
  Der name eines musters gibt die leistung in der knappstmöglichen form an. „Klassifikation“ bezieht sich auf eine einteilung der muster in drei kategorien: Erzeugungsmuster (*creational*), strukturmuster (*structural*) und verhaltensmuster (*behavioral*).
- **Absicht**
  Hier gibt es eine kurze textliche darstellung des problems, das durch das muster gelöst werden soll.
- **Alias**
  Falls das muster auch unter anderen bezeichnungen bekannt ist, werden diese hier aufgelistet.
- **Motivation**
  Hier wird eine situation geschildert, in der das muster sinnvoll angewendet werden kann und es wird aufgezeigt, in welcher weise das muster hier zur lösung beiträgt.
- **Anwendbarkeit**
  In welchen situationen kann das entwurfsmuster angewandt werden? Welche beispiele von schlechtem entwurf können dadurch vermieden werden? Wie erkennt man solche situationen?

- **Struktur**
  Die struktur der an dem muster beteiligten klassen wird (wie gesagt: nicht in UML!) grafisch dargestellt.

- **Teilnehmer**
  Dieser abschnitt enthält die klassen und/oder objekte, die an dem muster beteiligt sind und deren zuständigkeiten.

- **Zusammenwirkungen** (*collaborations*)
  beschreiben die art, wie die teilnehmer des musters zusammenarbeiten.

- **Konsequenzen**
  Zum einen werden die vorteile, die sich aus der verwendung des musters ergeben, aufgeführt. Zum anderen müssen auch die auswirkungen auf das system (architektur, details des entwurfs und der implementierung) beschrieben werden: welche nachteile müssen ggf. in kauf genommen werden, welcher teil der systemstruktur kann unabhängig von dem muster modifiziert werden?

- **Implementierung**
  Hier werden hilfestellungen, warnungen und besondere implementierungstechniken diskutiert, ggf. auch sprachspezifische aspekte.

- **Beispielcode**
  Codefragmente (in C++ oder Smalltalk) illustrieren, wie das muster implementiert werden könnte.

- **Bekannte anwendungen**
  In diesem abschnitt wird an mindestens zwei beispielen aus unterschiedlichen anwendungsbereichen vorgeführt, wo dieses muster erwiesenermassen schon (erfolgreich) verwendet wurde.

- **Verwandte muster**
  Welche entwurfsmuster sind ähnlich zu dem hier vorgestellten? Worin bestehen die wesentlichen unterschieden? Mit welchen anderen mustern zusammen sollte dieses muster verwendet werden?

4.7 Das beobachter-muster

Damit die diskussion über die objektorientierten muster nicht so abstrakt bleibt, stellen wir ein wichtiges muster hier etwas detaillierter dar: das *observer*

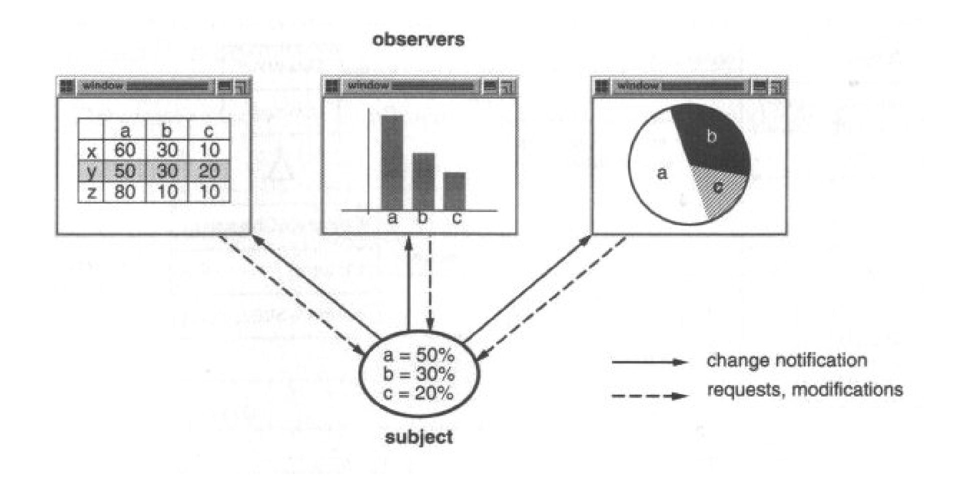


Abbildung 31: Gegenstand und beobachter

pattern. Dabei orientieren wir uns trotz aller kritik an Gamma u. a. (1995), wobei aber zahlreiche details unterdrückt werden. Das beobachter-muster präpariert einen wesentlichen aspekt des MVC heraus: dass es nämlich eine modellkomponente geben kann (hier *subject* genannt), die einen oder mehrere beobachter (*observer*) beeinflusst, s. abb. 31.

4.7.1 Name und klassifikation

Beobachter (Objekt-verhaltensmuster)

4.7.2 Absicht

Definiere eine $1 : n$-abhängigkeit zwischen objekten, so dass zustandsänderungen des einen objekts automatisch zu anpassungen in allen abhängigen objekten führen.

4.7.3 Alias

(Observer,) Dependents, Publish-Subscribe

4.7.4 Motivation

In ziemlich vielen systemen kommt es vor, dass eine reihe von objekten aufeinander synchronisiert werden muss, z. b. wenn ein bestimmter sachverhalt in unterschiedlichen fenstern auf verschiedene art angezeigt werden soll. Die

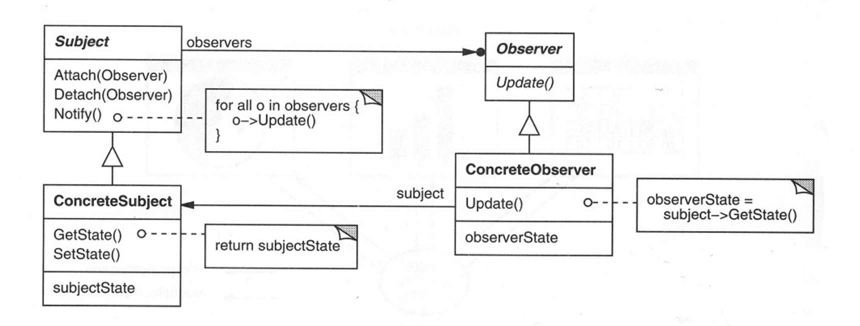
kopplung zwischen einer solchen ansammlung kooperierender klassen soll minimiert werden. Die schlüsselobjekte in diesem muster sind *gegenstand (subject)* und *beobachter (observer)*. Jeder gegenstand kann eine beliebige menge von ihm abhängiger beobachter haben. Das schema hier heisst auch *publish–subscribe*: Die beobachter registrieren sich bei einem gegenstand (*„subscribe"*); bei jeder änderung schickt der gegenstand dann benachrichtigungen (*„publish"*) an alle bei ihm registrierten beobachter, ohne dabei wissen zu müssen, wer genau diese beobachter sind. Auf eine solche benachrichtigung hin fragen die beobachter dann wiederum die für sie relevanten details beim gegenstand ab.

4.7.5 Anwendbarkeit

Dieses muster sollte immer dann angewendet werden, wenn

- eine abstraktion zwei aspekte hat, von denen einer von dem anderen abhängt,
- eine änderung in einem objekt änderungen in anderen objekten verlangt, ohne dass von vorneherein klar ist, wie viele objekte dies sein werden oder
- wenn ein objekt in der lage sein sollte, andere objekte zu benachrichtigen, ohne wissen zu müssen, welches diese objekte genau sind.

4.7.6 Struktur



(Die schrägschrift bei den oberen beiden klassen bedeutet, dass es sich hier um abstrakte klassen handelt. Beachte, dass gegenüber der UML hier die felder für attribute und methoden vertauscht sind!)

4.7.7 Teilnehmer

Subject (Gegenstand)

- kennt seine beobachter (Observer). Jede beliebige zahl von beobachtern kann einen gegenstand beobachten,
- hat eine schnittstelle, um beobachter-objekte anzumelden und abzumelden.

Observer (Beobachter)

- definiert eine `Update`-schnittstelle für objekte, die über eine änderung im gegenstand benachrichtigt werden sollten.

ConcreteSubject

- speichert einen zustand, der für `ConcreteObserver`-objekte von interesse ist,
- schickt seinen beobachtern eine nachricht, wenn sein zustand sich ändert.

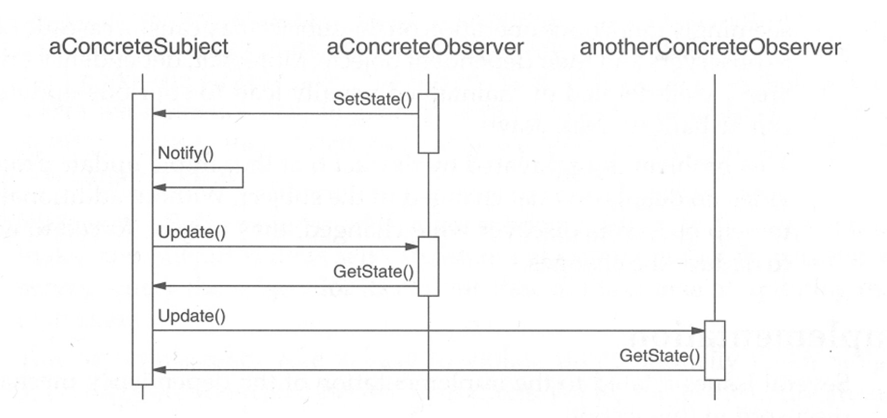
ConcreteObserver

- hält eine referenz auf ein `ConcreteSubject`-objekt,
- speichert einen zustand, der mit dem zustand des gegenstands konsistent bleiben sollte,
- implementiert die `Update`-schnittstelle des beobachters, um seinen zustand mit dem des gegenstands konsistent zu halten.

4.7.8 Zusammenwirkungen

- `ConcreteSubject` benachrichtigt seine beobachter, sobald eine änderung eintritt, die den internen zustand der beobachter inkonsistent mit seinem eigenen machen könnte.
- Nachdem ein `ConcreteObserver` von einer änderung benachrichtigt wurde, kann er beim gegenstand informationen abfragen und sie dazu verwenden, seinen zustand dem des gegenstands anzupassen.

Das folgende interaktionsdiagramm zeigt das zusammenwirken zwischen einem gegenstand und zwei beobachtern.



In diesem bild wartet der beobachter, der die änderung im gegenstand hervorgerufen hat, mit der anpassung seines eigenen zustands, bis er vom gegenstand dazu aufgefordert wird, und der gegenstand sendet sich selbst eine `Notify`-nachricht. Das muss nicht unbedingt so gemacht werden; es ist auch möglich, dass beobachter die `Notify`-nachricht senden.

4.7.9 Konsequenzen

Dieses muster erlaubt es, gegenstände und ihre beobachter separat zu ändern: beobachter können hinzugefügt und entfernt werden, ohne dass der gegenstand oder andere beobachter geändert werden müssen. Weitere konsequenzen sind:

1. Es gibt nur eine lose kopplung zwischen gegenstand und beobachter. Alles was ein gegenstand wissen muss, ist eine liste seiner beobachter, die alle auf eine einheitliche schnittstelle passen.

2. Das muster ist vorgerüstet für eine kommunikation nach dem *broadcast*-prinzip: Die benachrichtigungen, die von einem gegenstand ausgehen, müssen nicht zwangsweise an bestimmte beobachter gerichtet sein, sie können auch einfach zentral publiziert und von jedem „gehört“ werden, der daran interessiert ist.

3. Eine gefahr besteht darin, dass beobachter nicht wissen, wie viele andere beobachter auf den gegenstand registriert sind und wie teuer daher eine änderung des zustands im gegenstand sind. Unter umständen kann eine harmlose, von einem beobachter angeforderte zustandsänderung des gegenstands eine lange, kostspielige kette von zustandsänderungen in nachgeordneten beobachtern nach sich ziehen.

4.7.10 Implementierung

Bei der implementierung sind die folgenden fragen zu bedenken:

1. *Abbildung von gegenständen auf ihre beobachter.* Die einfachste methode der führung einer liste von beobachtern eines gegenstands ist es, die referenzen aller beobachter im gegenstand selbst abzuspeichern. Wenn es aber viele gegenstände gibt und nur wenige beobachter, könnte das zu viel speicher verbrauchen. Dann liesse sich die zuordnung auch z. B. in einer *hash*-tabelle speichern, was den zugriff auf die beobachter allerdings verlangsamt.

2. *Mehr als einen gegenstand beobachten.* Unter umständen beobachtet ein beobachter mehr als einen gegenstand. Dann muss die `Update`-methode so erweitert werden, dass der beobachter erkennen kann, welcher der gegenstände eine zustandsänderung hatte.

3. *Wer löst die anpassungen aus?* Für das senden der `Notify`-nachricht nach einer änderung des gegenstands gibt es im prinzip zwei möglichkeiten:

   (a) Der gegenstand sendet selbst `Notify` nach jeder von einem beobachter angeforderten zustandsänderung. Vorteil ist, dass sich die beobachter nicht um dieses detail kümmern müssen, nachteil ist, dass eine zusammengehörige folge von zustandsänderungen, die nur in ihrer gesamtheit ein `Notify` verlangen würden, zu einer möglicherweise unübersehbaren folge von `Update`-operationen in allen beobachtern führen würde.

   (b) Jeder beobachter sendet `Notify` nach einer relevanten zustandsänderung. Hier ist der vorteil, dass dies erst am ende einer zusammengehörenden folge von änderungen geschehen muss; der nachteil ist, dass es möglicherweise vergessen wird und dann zu inkonsistenten zuständen führt.

4. *Hängende referenzen.* Wenn ein gegenstand gelöscht wird, sollten seine referenzen in allen beobachtern auch gelöscht werden. Die beobachter selbst können nicht ohne weiteres gelöscht werden, da sie möglicherweise auch noch andere gegenstände beobachten.

5. *Garantie eines konsistenten zustands.* Bevor ein gegenstand bei allen seinen beobachtern die `Update`-methode aufruft, sollte er zuerst seinen inneren zustand konsolidieren. Auch wenn dies trivial erscheint, gibt es doch trickreiche situationen, wenn eine unterklasse eines gegenstands ererbte methoden aufruft.

6. *Push-modell vs. pull-modell.* Wie sollen beobachter an die für sie relevanten informationen kommen? Im push-modell schickt der gegenstand mit seiner `Update`-nachricht sofort alle informationen mit, auf die gefahr hin, dass manche beobachter gar nichts damit anfangen können. Im reinen pull-modell gibt es nur die nachricht, dass sich irgendetwas geändert hat und die beobachter müssen alles einzeln erfragen. Dazwischen sind natürlich allerlei abstufungen möglich.

7. *Explizite spezifikation von interessen.* Eine andere möglichkeit ist es, dass beobachter im zusammenhang mit ihrer registrierung beim gegenstand spezifizieren, für welche details im zustand sie sich interessieren. Dann kann der gegenstand seine `Update`-nachricht auf die beobachter beschränken, die sich für die geänderte zustandskomponente angemeldet haben.

8. *Kapselung komplexer `Update`-vorgänge.* Wenn die beziehungen zwischen den zuständen von gegenstand und beobachter sehr komplex sind, kann es sich lohnen, einen `ChangeManager` als instanz des *Mediator*-musters zwischenzuschalten.

4.7.11 Beispielcode

(Den schenken wir uns hier.)

4.7.12 Bekannte anwendungen

Das muster wird in zahlreichen situationen vor allem im zusammenhang mit grafischen benutzerschnittstellen verwendet; am bekanntesten ist das bereits erwähnte Model-View-Controller-muster von Smalltalk.

4.7.13 Verwandte muster

Das *Mediator*-muster kann verwendet werden, um komplexe `Update`-prozesse zu entkoppeln. Dabei kann auch das *Singleton*-muster verwendet werden, um einen `ChangeManager` eindeutig global verfügbar zu machen.

TODO: Jetzt nochmals kommentieren: vorkommen des observer pattern im MVC, auch die mögliche observer-beziehung zwischen controller und model.

TODO: weitere patterns: view handler, command, command processor, memento, layers, pipes & filters

4.8 Verteilte objektorientierte anwendungen

Gerade im OO-sektor hat sich das *client-server*-paradigma sehr etabliert, bei dem anbieter von diensten (*server*) und abnehmer hiervon (*client*) unter umständen auf räumlich getrennten maschinen eines netzwerks laufen. Konsequentes fortdenken dieses ansatzes bringt sofort den gedanken ein, dass *client* und *server* auch verschiedene betriebssysteme haben und in verschiedenen programmiersprachen programmiert sein können. Es wird dann eine gewisse „vermittlungsfunktion" („maklerdienst") benötigt, um einerseits die gewünschten dienste im netzwerk lokalisieren zu können und andererseits die unterschiedlichen aufrufkonventionen, datendarstellungen etc. aufeinander abzubilden.

4.9 Komponenten und frameworks

Viele autoren im dunstkreis objektorientierter techniken machen sich gedanken um die „zeit nach OO". Vieldiskutierte ansätze in diesem zusammenhang sind *komponenten* und *frameworks*. Was komponenten wirklich sind, bzw. inwiefern sie sich von modulen unterscheiden, ist nicht so ganz klar, ausser dass komponenten irgendwie „grösser" als module sind. Der „UML User Guide" (Booch 1998) sagt dazu

component „a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces".

Etwas klarer ist die situation in bezug auf frameworks. Zwar sagt der UML User Guide auch hierzu nur

framework „an architectural pattern that provides an extensible template for applications within a domain",

etwas genauer beschreiben aber Fayad und Schmidt (1997)

> *A framework is a semi-complete application that can be specialized to produce custom applications. In contrast to earlier OO reuse techniques based on class libraries, frameworks are targeted for particular business units (such as data processing or cellular communications) and application domains (such as user interfaces or real-time avionics).*

Frameworks sind wiederverwendbare, sozusagen halbfertige anwendungen, die sich leicht zu vollständigen programmen konfigurieren lassen. Sie stammen eindeutig aus dem bereich der graphischen benutzerschnittstellen

(z. B. MacApp, Qt, Microsoft MFC, X Windows) und gehen erst in neuerer zeit verstärkt auf vorfabrizierte anwendungen zu, etwa informationssysteme, finanztransaktionen, buchungssysteme.

Entsprechend ihres anwendungsgebiets können nach Fayad und Schmidt (1997) frameworks in drei unterschiedliche kategorien eingeteilt werden:

1. *System Infrastructure Frameworks*
   Diese frameworks dienen zur entwicklung der infrastruktur von systemen. Beispiele für diese art von frameworks sind betriebssystemzusätze oder kommunikations-frameworks.

2. *Middleware Integration Frameworks*
   Die integration verteilter komponenten wird durch *Middleware Integration Frameworks* unterstützt. Diese dienen dazu, modulare und wiederverwendbare software zu entwickeln, die in einer verteilten umgebung arbeiten soll. Zu diesen frameworks gehören beispielsweise Object Request Broker (ORB) oder transaktionsorientierte objektorientierte datenbanken.

3. *Enterprise Application Frameworks*
   Diese art von frameworks unterstützt die entwicklung von applikationssoftware, die direkt von einem endanwender genutzt wird. Damit unterscheiden sie sich von den beiden anderen typen von frameworks, die lediglich eine basis für applikationen bieten. *Enterprise Application Frameworks* werden unter anderem bei der entwicklung von software im kaufmännischen bereich, der produktionsplanung und -steuerung, für die telekommunikation oder der steuerung von automatisierungsgeräten im industriellen umfeld verwendet.

Typisch für alle frameworks ist das sogenannte *Hollywood-prinzip*[19] der „callbacks“: Im gegensatz zu einer modulbibliothek (s. abb. 32), welche funktionen anbietet, die vom klienten aufgerufen werden können, behält ein solches framework selbst die oberhand, und der klient kann lediglich bestimmte eigene funktionen anmelden, die vom framework unter bestimmten voraussetzungen aufgerufen werden. (Abb. 33).

Kontrollfragen

1. Was versteht man unter einem objekt?

[19] Den filmbossen in Hollywood wird nachgesagt, dass sie unerwünschte möchtegern-stars damit abwimmeln, dass sie sich eine visitenkarte geben lassen und dazu bemerken: „Don't call us, we'll call you“.

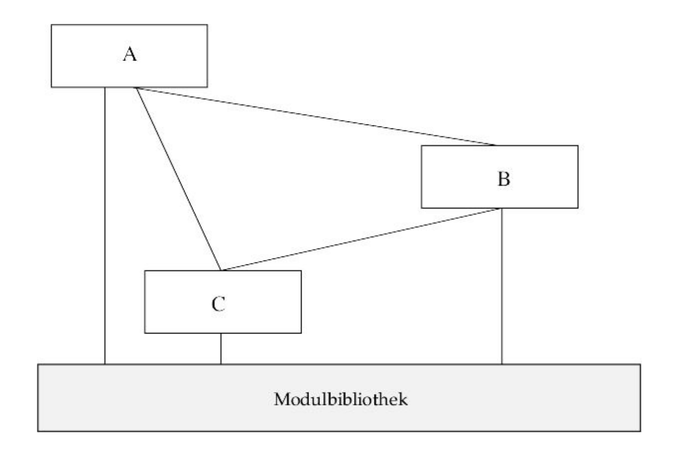


Abbildung 32: Modulares system

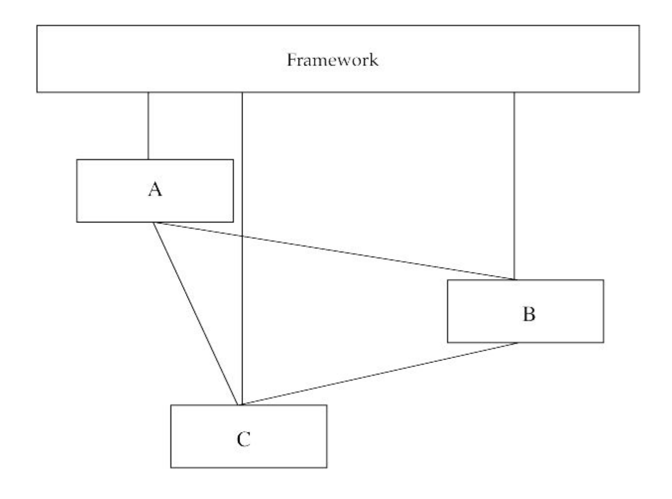


Abbildung 33: Framework

2. Was bedeutet objektidentität? Was ist eine objektreferenz? Wieso ist eine objektreferenz etwas anderes als ein zeiger?
3. Erklären sie gemeinsamkeiten und unterschiede zwischen den begriffen „modul“, „abstrakter datentyp“ und „klasse“.
4. Was versteht man unter dynamischer bindung?
5. Welche ansätze bietet die UML zur statischen modellierung von systemen?
6. Was versteht man unter einem muster? Welches sind die wesentlichen komponenten seiner beschreibung?
7. Beschreiben sie das beobachtermuster!
8. Beschreiben sie den unterschied zwischen modulbibliotheken und frameworks!

5 Softwarequalität

> *Testing, probably more than any other activity in software development, is about discovery. (Armour 2005)*
>
> *The fundamental law of bug finding is: No Check = No Bug. (Bessey u. a. 2010)*
>
> *It is probably true that algorithms have specifications, but this isn't true very often of "regular" code, which has been constructed by a process of piecemeal growth. (Gabriel 1997)*
>
> *I've bee in industry for over 10 years, and I don't see a whole lot of people proving correctness. (Gabriel 1997)*

In der theoretischen informatik sprechen wir von der *korrektheit* der software als dem höchsten gut und davon, dass diese korrektheit gegenüber einer formalen spezifikation des programms *bewiesen* werden muss. Diese ansicht wird z. b. sehr energisch von Edsger W. Dijkstra vertreten, der von einer *brandmauer* spricht, welche die ordentliche, strukturierte, formale welt der informatik abtrennt von dem ihm umgebenden urwald, der „real existierenden[20] welt." Softwareentwicklung geht dann davon aus, dass uns jemand eine formale spezifikation über die mauer wirft, diese prüfen wir informatiker dann zunächst auf widerspruchsfreiheit und machen uns sodann auf, ein programm zu entwickeln, das dieser spezifikation entspricht. Wir beweisen diese eigenschaft sogar formal, und dann werfen wir das fertige programm wieder über die mauer in den urwald. Wenn es dem empfänger nicht *gefällt* (Dijkstra spricht in der tat von *pleasantness* im gegensatz zur korrektheit), dann hat er die falsche spezifikation abgegeben.

Geht man davon aus, dass es in der tat möglich ist, eine vernünftige, zutreffende spezifikation zu schreiben, so ist sicherlich etwas wie „korrektheit" die wesentlichste eigenschaft von software, aber im gegensatz zur theoretikermeinung handelt es sich dabei nicht um eine boolesche variable $b \in \{T, F\}$, sondern um ein wahrscheinlichkeitsmass $p \in [0, 1)$.

In der praxis liegen die dinge (wie üblich) komplizierter[21]. Über die korrektheit hinaus sind viele andere eigenschaften von software relevant; In einem sehr lesenswerten artikel über haftungsfragen im zusammenhang mit software stellen Voas u. a. (1997) fest: „Korrekt ist unter umständen nicht gut genug." Damit zielen die autoren auf ein altbekanntes thema ab: es ist unter umständen gar nicht möglich, eine zutreffende spezifikation zu erstellen. Richard Gabriel

[20] Diese begriffsbildung ist nicht von Dijkstra, sondern ich habe sie hier eingeführt. Ob die schöpfer des begriffs „real existierender sozialismus", woher ich diese formulierung genommen habe, wohl auch eine gewisse ironie darin ausdrücken wollten? Jedenfalls wussten sie genau, dass ihr eigentliches ziel (der kommunismus) noch unvollkommen realisiert war; in der zwischenzeit musste die welt mit dem real existierenden sozialismus vorlieb nehmen.

[21] „Der hauptunterschied zwischen theorie und praxis liegt darin, dass es ihn in der theorie gar nicht gibt, in der praxis aber sehr wohl".

(1997) bringt — vor allem in dem sehr lesenswerten kapitel „Habitability and Piecemeal Growth“ — den gedanken ins spiel, gute software müsse *bewohnbar* sein und erklärt dies mit einem vergleich: wie kommt es, dass über lange zeiträume nach und nach gewachsene gebäude wie z. b. mittelalterliche burgen und klöster so „gemütlich“ wirken und moderne wolkenkratzer[22] so kalt und abweisend? Er zitiert hier wieder Christopher Alexander, der den begriff der *organischen ordnung* eingeführt hat, die er folgendermassen definiert:

> *die art von ordnung, die erreicht wird, wenn es einen perfekten ausgleich zwischen den bedürfnissen der teile und den bedürfnissen des ganzen gibt.*

Alexander spricht in diesem zusammenhang von der „qualität ohne namen“, die sich einstellt, wenn eine solche organische ordnung vorliegt. Gabriel (1997) listet unter anderem die folgenden eigenschaften von software auf, welche die qualität ohne namen besitzt:

- Wenn ich irgendeinen kleinen teil davon anschaue, kann ich sehen, was vor sich geht — ich brauche mich nicht auf andere teile zu beziehen, um zu verstehen, was irgendetwas tut. Dies sagt mir, dass die abstraktionen für sich allein sinn ergeben: sie sind ganzheitlich.
- Wenn ich irgendeinen grossen teil im überblick anschaue, kann ich sehen was vor sich geht — ich brauche nicht alle details zu wissen, um das zu begreifen.
- Sie ist wie ein fraktal, in dem jedes detail genau so lokal kohärent und so gut durchdacht ist wie jede andere ebene.
- Jeder teil des code ist transparent klar — es gibt keine abschnitte die dunkel sind, um effizienz zu gewinnen.
- Alles daran kommt mir vertraut vor.

Kommen wir nunmehr wieder auf den gesichtspunkt der (pseudo-) korrektheit von software zurück bzw. zu dem für die softwaretechnik wichtigeren thema der entdeckung und behebung von fehlern. Boehm und Basili (2001) listen die zehn wichtigsten erkenntnisse zur reduzierung von fehlern in software wie folgt auf:

1. *Ein softwareproblem nach der auslieferung zu finden und zu beheben, ist oft 100 mal teurer, als es während der anforderungs- und entwurfsphase zu finden und zu beheben*. Anhand empirischer daten aus zahlreichen projekten hat Boehm

[22]oder die DDR-plattenbauten, die Gabriel nicht bekannt waren

ermittelt, daß die kosten für die beseitigung eines fehlers bei der systemanalyse exponentiell mit dem abstand von dieser frühen phase steigen. Boehm und Basili betonen aber, dass es bei kleineren softwaresystemen häufig nur ein faktor 5 ist.

2. *Heutige softwareprojekte verwenden rund 40 bis 50 prozent ihres aufwands auf vermeidbare überarbeitungen*. Als *vermeidbar* werden dabei überarbeitungen definiert, sie sich bei einer besseren analyse des problems nicht ergeben hätten. Unvermeidbar sind demgegenüber überarbeitungen, die sich aus nachträglichen änderungen der anforderungen ergeben.

3. *Ungefähr 80 prozent der vermeidbaren überarbeitungen gehen auf 20 prozent der fehler zurück*[23].

4. *Ungefähr 80 prozent der fehler stammen aus 20 prozent der module und ungefähr die hälfte der module sind fehlerfrei.* Mehrjährige messdaten zeigen prozentsätze zwischen 60 und 90 prozent mit einem median bei 80.

5. *Ungefähr 90 prozent der ausfallzeiten*[24] *geht auf 10 prozent der fehler zurück.* Daten aus neun grossen IBM-projekten zeigen sogar auf, dass 90 prozent der fehler auf rund 0,3 prozent der fehler zurückzuführen sind.

6. *Inspektionen durch kollegen finden 60 prozent der fehler*. In mehrjährigen messdaten fanden sich prozentsätze zwischen 31 und 93 prozent mit einem median von 60.

7. *Perspektivenbasierte inspektionen finden 35 prozent mehr fehler als ungerichtete inspektionen*. Hierbei geht es darum, inspektionen nicht allgemein und losgelöst von spezifischen problemen durchzuführen („Könnte irgend etwas an diesem code nicht in ordnung sein?“), sondern gezielt bestimmte für die anwendung bzw. den auftraggeber wichtige szenarien — unter umständen auf der basis der erfahrung mit älteren fehlerfällen — durchzuspielen.

8. *Disziplinierte persönliche praktiken können die fehler-einführungsraten um bis zu 75 prozent senken*. Zu solchen disziplinierten praktiken gehört beispielsweise das von Linger (1994) beschriebene *Cleanroom Process Model* oder der *Personal Software Process* von Humphrey (2000).

[23] Solche verteilungen nennt man *Pareto-verteilungen* nach dem italienischen volkswirtschaftler und soziologen Vilfredo Pareto (1848–1923); sie finden sich sehr häufig in allen möglichen bereichen. Siehe dazu unbedingt das interessante buch von Koch (1998)

[24] Ausfallzeiten können richtig teuer werden. Von Reinhard Bündgen (IBM) hörte ich, dass eine stunde ausfall bei einem geldautomatendienst 14.500 USD kostet, bei einem flugreservierungsdienst 89.500 USD, bei einer kreditkarten-authentisierung 2.600.000 USD.

9. *Wenn alle anderen parameter übereinstimmen, dann kostet es 50 prozent mehr pro quellinstruktion, hochzuverlässige softwareprodukte zu entwickeln als softwareprodukte mit niedriger zuverlässigkeit. Andererseits ist diese investition mehr als gerechtfertigt, wenn das projekt spürbare betriebs- und wartungskosten beinhaltet.*

10. *Ungefähr 40 bis 50 prozent der benutzerprogramme enthalten nicht-triviale fehler*. Hier geht es um „programme", die der *benutzer* eines softwareprodukts (z. b. eines tabellenkalkulationsprogramms) selbst schreibt. Die lehre aus dieser beobachtung ist, dass entwickler von programmierbaren anwendungen genügend viele „sicherheitsgurte und airbags" in ihr produkt einbauen müssen.

Eine übliche methode, sich von der korrektheit von software zu überzeugen, ist das testen. Von E. W. Dijkstra stammt die bemerkung, dass wir durch testen stets nur die *anwesenheit* von fehlern nachweisen können, aber nicht deren abwesenheit. Vom standpunkt eines theoretikers ist dies vollkommen richtig; in der praxis gibt es jedoch verfahren, die recht präzise statistische aussagen ermöglichen, wie viele fehler in einem programm noch zu erwarten sind.

Graham (2002) listet irrmeinungen zum testen auf, die beachtung verdienen. Auch wenn sich diese irrmeinungen zum teil gegenseitig widersprechen, finden sie sich doch alle irgendwo in der literatur wieder:

1. **Falsch: „Anforderungen stehen am anfang, tests am ende."** In wirklichkeit können auftraggeber, die bereits während der anforderungsanalyse aufgefordert werden, testfälle (inklusive der erwarteten ergebnisse) für das zu erstellende softwareprodukt zu liefern, dadurch frühzeitig missverständnisse ausräumen. Die moderne XP-methode der softwareentwicklung erhebt dies mit ihrem *test driven development* sogar zum prinzip. Umgekehrt können auch anforderungsdokumente bereits getestet werden.

2. **Falsch: „Testen kann man erst, wenn das system fertig ist."** In abschnitt 5.5 werden wir darlegen, dass die weitaus grösste zahl der fehler bereits in frühen anforderungs- und entwurfsdokumenten gefunden werden kann.

3. **Falsch: „Anforderungen braucht man zum testen, aber nicht umgekehrt."** Natürlich wird ein system in der regel gegen (! beachte die terminologie!) ein anforderungsdokument getestet. Bei unklaren oder nicht vorhandenen anforderungen kann man nicht testen. Aber, wie bereits ausgeführt, kann die qualität von anforderungsdokumenten verbessert werden, wenn frühzeitig der gedanke ans testen eingebracht wird.

4. **Falsch: „Wenn es schwierig ist, testfälle zu finden, dann ist das nur ein problem des testens.“** Leider sind nicht wirklich alle anforderungen so formuliert, dass sie auch testbar sind. Es kann helfen, bereits bei der anforderungsdefinition darauf zu achten, dass nur testbare anforderungen festgeschrieben werden. „Testbar“ heisst in diesem zusammenhang in jedem fall auch „messbar“. Anforderungen wie „leicht zu benutzen“, „benutzerfreundlich“, „sehr zuverlässig“ oder „akzeptabler durchsatz“ sind unbrauchbar. Graham zitiert Gilb (1988) (s.a. s. 241), der gesagt hat: „Alles kann messbar gemacht werden in einem ausmass, das dem nicht-messen überlegen ist.“

5. **Falsch: „Kleinere änderungen in den anforderungen haben wenig auswirkungen auf das projekt.“** Software ist digital und deshalb zu einem maximalen grad unstetig. Es gibt keine „kleineren änderungen“. Selbst scheinbar kleinste änderungen verdienen zumindest einen regressionstest, ob alle testfälle noch problemlos durchlaufen.

6. **Falsch: „Zum testen braucht man die anforderungsdefinition nicht wirklich.“** Ein testfall beinhaltet eingaben, vorbedingungen an die eingaben und erwartete ausgaben. Nur die anforderungsdefinition kann hierzu fundierte angaben liefern.

7. **Falsch: „Testen kann man nicht ohne ein anforderungsdokument.“** Es gibt einen begriff des „explorativen testens“, bei dem ein system weniger echt getestet als vielmehr erkundet wird. Trotzdem kann ein intelligenter tester auch hier bereits eine fülle von ungereimtheiten aufdecken und zumindest interessante fragen aufwerfen, die zu beantworten wären. Falsch ist ebenfalls die (in dem o .a. satz möglicherweise auch enthaltene) auffassung, dass der tester erst dann mit seiner arbeit beginnen kann, wenn das anforderungsdokument fixiert ist. Vernünftige testfälle können vielmehr ein wichtiger bestandteil eines anforderungsdokuments sein, und tester können dabei mithelfen, diese zu formulieren.

Ein weitverbreiteter irrtum in bezug auf fehler in softwareprodukten ist es, dass software ohne bzw. mit wenig fehlern viel teurer ist als fehlerhafte software. Sneed und Jungmayr (2011) zitieren, dass

> *„die Behebung von Fehlern zwischen 25 und 40 % der gesamten Wartungskosten ausmacht, die wiederum für mindestens 67 % der Lebenszykluskosten verantwortlich sind. Demnach macht die Fehlerkorrektur 17–26 % der gesamten Lebenszykluskosten aus.“*

Sie weisen in der folge noch darauf hin, dass ausserdem ja auch die durch fehler beim kunden verursachten folgekosten recht erheblich sein können. Als

beispiel führen sie das *Arbeitslosenunterstützungssystem ALU-II* vor, das 25 millionen Euro zu viel auszahlte. Eine parlamentarische untersuchung des landes Schleswig-Holstein zeigte, dass eine manuelle auszahlung der arbeitslosengelder in einem bundesland jährlich 300 millionen Euro gekostet hätte, durch die fehler im ALU-II-System kostete sie jedoch 380 millionen Euro: wegen der fehler in der berechnung gingen täglich zwei arbeitsstunden pro sachbearbeiter verloren.

Viele aussagen im bereich der software-qualitätssicherung sind statistischer natur und deshalb mit den üblichen warnungen vor statistischen aussagen zu belegen. Die meisten statistischen eigenschaften einer gesamtheit von objekten können an einem einzelfall oft in dramatischer weise widerlegt werden.

Vieles von dem zahlenmaterial stammt noch aus der zeit der grossrechner, deren relativ hoher preis und dementsprechend relativ niedrige installationshäufigkeit eine genaue verfolgung von fehlern möglich machte. Sehr ausgefeilt ist das von Adams (1984) geschilderte verfahren bei der firma IBM, das aus der grossrechnerzeit (mit eher geringen anzahlen installierter maschinen) stammt und jetzt auch auf mittlere maschinen und workstations mit ungleich höheren installationzahlen übertragen wurde. Vermutet hier ein kunde einen software-fehler, so benachrichtigt er eine zentrale problemaufnahmestelle, die einen „*Problem Management Record*“ (PMR) anfertigt. Ein level-1-support team prüft anhand entsprechender datenbanken, ob dieses problem schon einmal aufgetreten ist und teilt dem kunden ggf. eine lösung dafür mit. Anderenfalls wird der level-2-support eingeschaltet, der unter umständen auf das *change team* beim entwicklungsteam zurückgreift. Wenn es sich um ein neues softwareproblem handelt, wird dort ein APAR („*Authorized program analysis report*“) erstellt. Je nach dringlichkeit des problems wird dann entweder spezifisch für diesen kunden ein „*code change*“ hergestellt, der sein problem behebt oder mit etwas grösserem aufwand ein PTF („*program temporary fix*“) hergestellt, der allen benutzern des gleichen programms zugute kommt. Natürlich werden die PTFs auch der für die entwicklung der nächsten software-ausgabe zuständigen stelle mitgeteilt. Ungefähr im monatsabstand werden dann PTF-bänder an alle kunden versandt[25], damit diese entweder korrigierende wartung (CS=„*corrective service*“) für bereits bei ihnen aufgetretene fehler oder aber vorbeugende wartung (PS=„*preventive service*“) für noch nicht aufgetretene fehler durchführen können. Der ursprüngliche PMR wird erst dann zu den akten gelegt, wenn der kunde bestätigt hat, dass sein problem gelöst wurde und dass er mit dieser lösung zufrieden ist.

In der praxis führt dies häufig dazu, dass die supportstellen erst einmal nachfragen, ob alle versandten PTFs auch installiert sind, bevor sie sich mit

[25]weswegen einige IBM-kunden die abkürzung PTF als „*permanent temporary fix*“ bezeichnen

einem problem ernsthaft beschäftigen. Andererseits machten auch einige kunden, die eilfertig alle zugesandten PTFs sofort installierten, schon die trübe erfahrung, dass durch die PTFs plötzlich bei ihnen erst fehler auftraten, wo vorher alles reibungslos verlaufen war. Dies führte natürlich zu einer gewissen abneigung gegen PTFs, zumal das komplexe geflecht der abhängigkeiten („PTF 40190 setzt PTF 38175 voraus. . . “) zwischen PTFs nicht immer zu durchblicken ist.

Die datenbanken über APARs und PTFs sind übrigens auch für die kunden direkt im internet einsehbar.

In der zitierten arbeit von Adams (1984) wird deshalb mit statistischen methoden eine abschätzung versucht, wann sich ein PS gegenüber einem CS wirtschaftlich rechnet. Gleichzeitig ergeben sich dabei interessante daten über die fehlerhäufigkeit und -anzahl (s. abb. 34; die zugrundeliegenden daten sind auch in tab. 4 aufgeführt). Die aufbereitung der daten stammt von Cobb und Mills (1990), woher auch die bodenzeile aus tab. 4 stammt. Ich habe jedoch die acht separaten MTTF-klassen[26] in sechs zusammengefasst. Wir wollen an

MTTF-Klasse	≥1583a	500a	158a	50a	15.83a	≤5a
Prozent der Fehler	62,3%	18,2%	9,7%	4,5%	3,2%	2,2%
Wahrscheinlichkeit	2,9%	4,4%	7,9%	12,3%	18,7%	53,7%

Tabelle 4: MTTF nach Adams

dieser stelle nicht diskutieren, wie Adams anhand der aufzeichnungen über APARs und PTFs zu den genannten zahlen gelangt ist. Hier finden komplizierte normierungen bezüglich der anzahl der benutzer auf einer maschine, der geschwindigkeit der maschine und anderen faktoren statt, um letzten endes die MTTF für neun weithin genutzte software-komponenten von IBM zu ermitteln. Aus tab. 4 ergibt sich für den durchschnitt der neun ausgewählten software-produkte, die sich im übrigen bemerkenswert gleichartig verhalten, dass die überwiegende anzahl der fehler eine MTTF von mindestens 1583 jahren haben, das heisst, bezogen auf einen einzelnen benutzer tritt dieser fehler in 1583 jahren höchstens einmal auf. Nur 2,2% der fehler liegen in der niedrigsten fehlerklasse, wo sie eine MTTF von bis zu 5 jahren haben. Anders formuliert tritt ein solcher fehler täglich einmal auf, wenn 10.000 benutzer das entsprechende programm vier stunden am tag einsetzen. Dies sind die wirklich wichtigen fehler, deren beseitigung einen spürbaren positiven effekt auf die softwarequalität hat. Fast zwei drittel der fehler dagegen sind so selten, dass sie in der regel kaum als solche wahrgenommen werden. Bei den oben angenommenen 10.000 benutzern mit jeweils vierstündigem programmeinsatz an

[26]MTTF = *Mean time to failure*

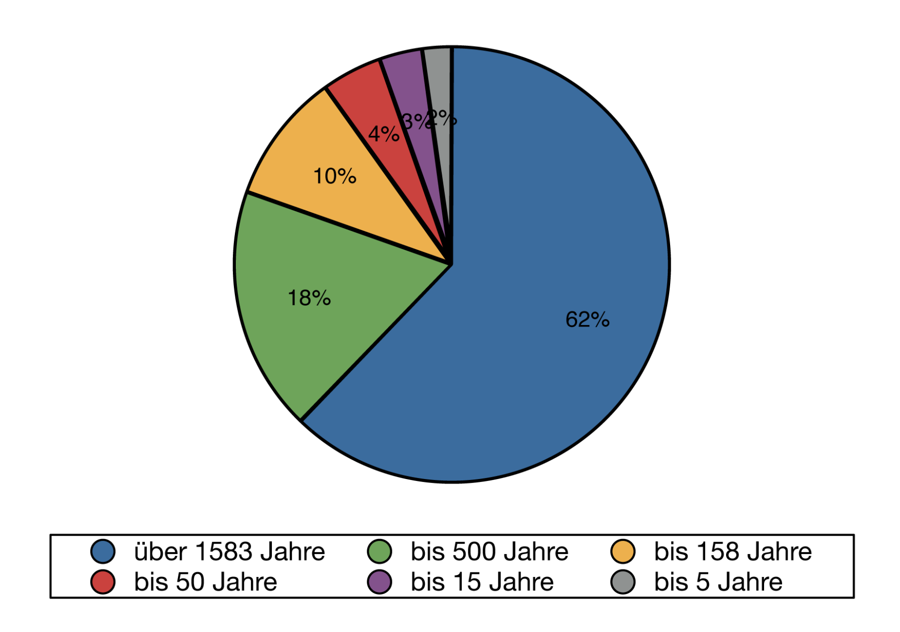


Abbildung 34: Fehlerhäufigkeit nach Adams

einem tag tritt ein fehler mit einer MTTF von 1.583 jahren nur ungefähr einmal im Jahr auf. Die unterste zeile in tab. 4 listet die wahrscheinlichkeit auf, mit der ein von einem benutzer bemerkter fehler zu einer der darüberstehenden MTTF-klassen gehört. Demnach gehören 62,3% der beseitigten fehler zu einer fehlerklasse, die der benutzer nur mit einer wahrscheinlichkeit von 2,9% entdeckt (genauer: 2,9% der vom benutzer beobachteten fehler gehören zu dieser fehlerklasse). Auf der anderen seite gehören nur 2,2% der beseitigten fehler zu einer fehlerklasse, die der benutzer mit mehr als 50%iger wahrscheinlichkeit beobachtet.

5.1 Value-driven testing

Sneed und Jungmayr (2011) stellen ausführliche betriebswirtschaftliche rechnungen zur wirtschaftlichkeit von tests an und berechnen unter anderem den *Test-ROI (Return on investment)*. Die testkosten werden hier aufgeteilt in *testressourcenkosten* und *testpersonalkosten* und hängen direkt ab von

- anzahl der testfälle,
- testbarkeitsfaktor und
- testautomatisierungsfaktor.

Nicht alle softwaresysteme sind gleich gut testbar, insbesondere wird die testbarkeit stark durch die verwendung von GUIs behindert. Auch die breite von export- und importschnittstellen und die anzahl von attributen in einer datenbank beeinflussen die testbarkeit. Wünschenswert ist es, möglichst viele tests zu automatisieren, aber auch bei einem vollautomatisierten test ist immer noch erfahrungsgemäss mindestens 25 % menschliche arbeit für die vorbereitung und die spätere kontrolle der auswertung notwendig, so dass der automatisierungsfaktor maximal 0,75 sein wird. Wir werden in abschnitt 5.7 nochmals auf den test-ROI zurückkommen.

5.2 Fehlerdichte

Wir messen häufig die *fehlerdichte* eines programms in fehlern pro kLOC. Es gibt untersuchungen von 1994, bei denen bis zu 25 fehler pro tausend programmzeilen gefunden wurden. Das klingt sehr viel, bedeutet aber, bezogen auf die programmzeilen, nur eine fehlerquote von 2,5%.

Sneed und Jungmayr (2011) zitieren aktuelle fehlerraten aus dem jahr 2006, die wir hier auszugsweise in tab. 5 wiedergeben.

Bereich	*mittl. fehler/kLOC*
Militär – Raumfahrt	0,4
Militär – Luftfahrt	0,5
Militär – Boden	0,8
Eingebettete Systeme	1,0
Automatisierungstechnik	5,0
Finanzwirtschaft	6,0
Telekommunikation (Switching)	6,0
IT-Systeme (Datenbanken)	8,0
Webapplikationen	11,0

Tabelle 5: Fehlerraten nach branche in den USA

Wirklich gute software hat heute etwa zwei bis drei fehler pro tausend zeilen, also eine fehlerquote von zwei bis drei promille. Es geht aber noch besser: Von einem hersteller medizinischer geräte weiss ich, dass dort nur ca. 2 fehler pro zehntausend zeilen gemacht werden[27]. Übrigens zeigt ein Buch von Holzmann (2003), dass erfahrene programmierer keineswegs weniger fehler machen als neulinge: die situation ist sogar noch schlimmer, weil die fehler, die erfahrene programmierer machen, sehr komplex und subtil sind; ihre entdeckung ist sehr schwierig und erfordert viel denkarbeit.

An dieser stelle mag man sich fragen, wie denn die verbleibenden fehler in einem softwareprodukt ermittelt werden. Letzten endes können wir im grunde doch nur die aus einem softwareprodukt entfernten fehler messen. Die antwort auf diese frage ist einfach:

1. Einerseits kann man als professioneller softwareentwickler seine erfahrung ins spiel bringen und die in einem bestimmten softwareprodukt im laufe seines lebens gefundene fehlerdichte in relation setzen zu einem ähnlichen und unter ähnlichen bedingungen zu entwickelnden softwareprodukt. Die im „alten“ produkt aufgefundene tatsächliche fehlerdichte kann als erwartungswert der fehlerdichte im „neuen“ produkt angesehen werden.

2. Wie bei allen solchen statistischen verfahren geht es nicht um die beurteilung einzelner softwarekomponenten, sondern ganzer herstellungsprozesse von software. Zur ermittlung der fehlerdichte bei einem bestimmten

[27]Dabei werden zwischen 20 und 185 stunden für das auffinden eines einzelnen fehlers investiert; die testzeit für 1kLOC beträgt zwischen 25 und 185 stunden; das bedeutet reine testkosten zwischen 1.660 und 12.300 EUR pro kLOC. Vor jedem software-release wird ein regressionstest gegen eine datenbank mit allen bekannten fehlern gefahren; dieser test braucht bei 900 kLOC C-code zwei wochen, 24h rund um die uhr.

herstellungsprozess kann man sich deshalb des sog. „*error seeding*“ bedienen: Basierend auf statistischen erfahrungsdaten werden in ein angenommenerweise fehlerfreies softwareprodukt ohne kenntnis der softwareentwickler eine typische anzahl typischer fehler eingebaut. Nach dem durchlaufen des normalen qualitätssicherungszyklus lässt sich dann *exakt* ermitteln, wieviel fehler in dem produkt verblieben sind; daraus ergibt sich dann eine *statistische* aussage, wieviel fehler pro kLOC ein typisches programm mit dieser qualitätssicherung haben wird.

Die erste methode wird normalerweise bessere resultate aufweisen, denn das *error seeding* wird unter anderem dadurch behindert, dass die fehler nicht gleichverteilt auftreten; dies wurde bereits zu eingang dieses kapitels in der liste von Boehm und Basili (2001) erwähnt. So hat es sich in einem bestimmten komplexen softwareprodukt der IBM herausgestellt, dass in einem gewissen zeitraum 2.133 von 2.731 modulen ohne fehler geblieben sind, während andererseits allein in einem einzigen modul 37 fehler gemeldet wurden. Ähnliches berichten Ostrand und Weyuker (2002) über ein grosses softwaresystem, das sie über einen längeren zeitraum beobachtet haben: dort finden sich 100% der fehler in nur 7% der beteiligten dateien, welche ihrerseits 24% der gesamten LOC beinhalten. Über mehrere änderungsstände der software hinweg ergibt sich ausserdem der effekt, dass die meisten fehler in den dateien enthalten sind, bei denen sich schon bei den ersten releases fehler zeigten. Sie schliessen deshalb: „Wir sehen deshalb hinweise darauf, dass es der mühe wert sein könnte, den aufwand für fehlerentdeckung und -beseitigung auf die relativ kleine anzahl fehleranfälliger dateien zu konzentrieren, wenn diese sich früh identifizieren liessen.“

Wie alle relativen masse ist fehler/kLOC etwas tückisch; Carey (1996) fragt zu recht:

> *Wenn wir einen fehler beseitigen, indem wir code hinzufügen, hat sich die softwarequalität verbessert? Ja. Wenn wir einen fehler beseitigen, indem wir code entfernen, hat sich die softwarequalität verbessert? Ja. Wenn wir einen fehler beseitigen, ohne code hinzuzufügen oder zu entfernen, hat sich die softwarequalität verbessert? Ja. Der gebrauch von fehler/kLOC widerspricht dem ursprünglichen ziel, die softwarequalität zu verbessern. [...] Es gibt einen punkt, wo wir, wenn wir genug code bei der fehlerbeseitigung entfernen, die intrinsische softwarequalität überhaupt nicht verbessern.*

$\frac{e-1}{k_0+k} \leq \frac{e}{k_0}$

$\frac{e-1}{k_0-k} \not\leq \frac{e}{k_0}$

$\frac{e-1}{k_0} \leq \frac{e}{k_0}$

5.3 Wo sind die fehler?

Ein interessantes problem ist natürlich, herauszufinden, *wo* sich in einem system die fehler finden lassen sollen. Wie bereits zuvor bemerkt, sind fehler niemals gleichverteilt, und es lohnt sich absolut nicht, an allen beliebigen stellen

nach fehlern zu suchen. Hierzu liefern Ostrand u. a. (2004) einen interessanten beitrag. Sie haben über ein grosses system der firma AT&T über 17 *releases* hinweg die aufgefundenen fehler sorgfältig untersucht und statistischen untersuchungen unterworfen.

Rel	# Dat	LOC	Fehler	Dichte	Fehlerh. dateien	% fehlerh. dateien
1	584	145.967	990	6,78	233	39,9
2	567	154.381	201	1,30	88	15,5
3	706	190.596	487	2,56	140	19,8
4	743	203.233	328	1,61	114	15,3
7	993	291.719	207	0,71	106	10,7
10	1372	396.209	246	0,62	112	8,2
17	1950	538.487	253	0,47	122	6,3

Tabelle 6: Fehlerverteilung bei Ostrand, Weyuker und Bell

In tab. 6 sind einige ihrer ergebnisse zusammengetragen. Dabei bedeutet

Rel die nummer des release

Dat die anzahl der dateien in dem *release*

LOC die gesamtzahl der *lines of code*

Fehler die gesamtzahl der gefundenen fehler

Dichte die fehlerdichte in fehlern pro 1kLOC

Fehlerh. dateien die anzahl der dateien mit mindestens einem fehler

% fehlerh. dateien den prozentsatz der dateien mit mindestens einem fehler.

Beobachtungen aus diesen daten sind zunächst einmal

1. Die grösse des systems ist über die zeit hinweg gewachsen, sowohl was die anzahl der beteiligten dateien betrifft als auch die gesamtzahl der *lines of code*. Das ist natürlich, da durch nachfolgende *releases* nicht nur fehler beseitigt, sondern auch neue funktionen hinzugefügt werden.

2. Die fehlerdichte lag bereits beim zweiten *release* unterhalb der hier erwähnten 2 fehler pro kLOC und ist von da an im wesentlichen gesunken bis zu einem halben fehler pro kLOC.

3. Waren zu beginn der entwicklung noch knapp 40% der dateien fehlerhaft, so waren es am ende nur noch gut 6%. Das bedeutet jedoch, dass knapp 94% der dateien überhaupt nicht mehr auf fehler untersucht werden müssen, da sie sowieso fehlerfrei sind!

Die wichtige frage in diesem zusammenhang ist natürlich, wie sich vorhersagen lässt, in welchen dateien die fehler mit grösster wahrscheinlichkeit anzutreffen sind. Dazu haben Ostrand et al. statistische untersuchungen durchgeführt, in die eine anzahl von variablen eingeflossen sind: der logarithmus der LOC, der änderungsstatus der datei (neu, geändert, ungeändert), alter der datei in anzahl von *releases*, quadratwurzel der im letzten *release* in dieser datei gefundenen fehler, programmiersprache und nummer des *release*. Vielleicht nicht überraschend, hat sich dabei herausgestellt, dass die wesentliche bestimmungsvariable die grösse der datei (in LOC) ist; diese allein ergibt eine zuverlässigkeit der vorhersage von 73%; das kompliziertere modell kann immerhin mit 83%-iger wahrscheinlichkeit die dateien anzeigen, in denen nach fehlern gesucht werden muss.

5.4 Dynamische analyse: testen

Sneed (1988) teilt testtechniken entsprechend einer von Miller und Howden vorgegebenen terminologie in zwei hauptkategorien ein: *statische analysemethoden* und *dynamische analysemethoden*. Nach heutiger terminologie würde ich die statischen analysemethoden eher zum thema „inspektionen“ zählen, das wir noch danach behandeln. Die meisten leute dürften unter „testen“ dynamische analysemethoden verstehen, bei denen man das programm tatsächlich laufen lässt. Dabei spielt der begriff des *testfalls* eine grosse bedeutung. Wir verstehen unter einem testfall zunächst einmal eine bestimmte kombination von eingabedaten; in zweiter linie dann aber auch die dazugehörigen ausgabedaten bzw. systemreaktionen.

Gemäss der zuvor erwähnten beobachtung, dass wir durch testen stets nur die *anwesenheit* von fehlern nachweisen können, aber nicht deren abwesenheit, sprechen wir von einem *erfolgreichen testfall* dann, wenn es gelungen ist, mit seiner hilfe einen fehler nachzuweisen; anderenfalls war der testfall erfolglos. Wir werden im folgenden drei grundsätzlich verschiedene dynamische testmethoden besprechen, die allesamt irgendwo ihre berechtigung haben:

Ablaufbezogenes testen: hier bildet der prozedurale ablauf die basis für die ermittlung der testfälle, d. h. ein testfall wird mit der absicht konstruiert, bestimmte abläufe im programm zu provozieren.

Datenbezogenes testen: hier werden testfälle auf der basis der datenbeschreibungen konstruiert. Meist wird dabei aber auf die überprüfung der ein-/ausgaberelation verzichtet.

Funktionsbezogenes testen: hier dient die funktionale beschreibung bzw. die spezifikation als basis für die ermittlung von testfällen.

Back-to-back testen hier werden zwei verschiedene versionen des programms in ihrem verhalten gegeneinander verglichen.

Ausserdem unterscheiden wir beim testen gerne die hauptkategorien

black box test Das quellprogramm wird nicht benötigt bzw. ist nicht verfügbar.

white box test Das quellprogramm wird zum test mitverwendet.

Bei manchen softwareentwicklern haben *white box tests* die höhere reputation; einer hat das in die worte gekleidet: „Das programm hat ein recht, angehört zu werden". Die später hier noch anzusprechenden inspektionsmethoden gehen *nur* von dem quelltext des programms aus, ohne es laufen zu lassen; wie sich zeigt, sind diese methoden in summa erfolgreicher als alle teste.

5.4.1 Ablaufbezogenes testen

Beim ablaufbezogenen testen (auch „*structured testing*" genannt) steht der ablaufgraph des zu testenden programms im vordergrund. Bei der auswahl von testfällen geht es darum, möglichst viele pfade durch das programm zu erzwingen. Man spricht hier von einer „pfadüberdeckung" (*coverage*). Dieser ansatz wurde erstmals 1975 von Miller vorgeschlagen, der ein werkzeug geschrieben hatte, welches einerseits testfälle mit dem ziel eines bestimmten überdeckungsgrades erzeugte und andererseits im laufenden programm die messung der tatsächlich angetroffenen überdeckung erlaubte. Anfang der 80er jahre war diese methode relativ populär, vor allem auch wegen ihrer bezüge zu graphentheoretischen eigenschaften des programmablaufgraphen, dessen komplexität durch die von McCabe (1976) vorgeschlagene zyklomatische komplexität gemessen werden kann.

Wir sprechen dann beispielsweise von einer 50%igen abdeckung, wenn 50% der in betracht gezogenen Pfade überdeckt sind, d. h. also während eines testfalls betreten werden. Die relevanz des ablaufbezogenen testens ist jedoch gering. Vergleiche etwa die empirischen daten von Adams (s. tab. 4): Wir werden bei hohen überdeckungsgraden möglicherweise viele fehler finden, die jedoch

in der praxis nie oder fast nie auftreten werden; andererseits ist vollkommen klar, dass wir mit ablaufbezogenem testen niemals alle fehlerarten entdecken können. Experimentelle daten zeigen (Sneed 1988), dass auch bei sehr hohen überdeckungsgraden stets weniger als 50% der aufgefundenen fehler durch ablaufbezogenes testen entdeckt wurden. Es lassen sich in der tat nämlich nur bestimmte arten von fehlern auf diese art und weise entdecken, z. b. unerreichbarer code, endlose schleifen und unvollständige bzw. widersprüchliche bedingungen. Da die spezifikation des programms völlig ausser acht bleibt, können logische fehler wie etwa vergessene funktionen, unberücksichtigte daten, inkonsistente verwendung von schnittstellen etc. auch bei höchster überdeckung niemals aufgefunden werden. Nach Sneed liegt der hauptnutzen des ablaufbezogenen testens darin, dass es den tester zwingt, sich intensiv mit dem testobjekt auseinanderzusetzen. Bei der suche nach testfällen zu einem gewissen überdeckungsgrad wird er bzw. das von ihm eingesetzte softwarewerkzeug mit grosser wahrscheinlichkeit auf logische ungereimtheiten und lücken im code stossen. Das ablaufbezogene testen ist vergleichbar mit dem testen einer eisfläche durch vorsichtiges betreten: wenn alle bereiche der fläche einmal betreten worden sind, kann man davon ausgehen, dass ein mensch mit dem gleichen oder einem niedrigeren gewicht in der näheren zukunft hier nirgendwo einbrechen wird. Über die genauen physikalischen eigenschaften der eisfläche ist damit noch nichts gesagt.

5.4.2 Datenbezogenes testen

Diese technik ist auch als „*volume testing*“ oder „*stress test*“ bekannt. Hier werden in grosser menge, gegebenenfalls auf der basis vorgegebener wahrscheinlichkeitsverteilungen, gültige kombinationen von eingabedaten zufallsmässig erzeugt und in das programm eingespeist. Aufgrund dieses zufallprinzips ist auch das jeweils zu erwartende ergebnis im prinzip unvorhersehbar. Der haupteinsatzzweck dieser technik liegt deshalb auch darin, möglichst programmabstürze zu provozieren und deren ursachen anschliessend zu ermitteln. Auch eine stichprobenartige überprüfung der ausgabedaten bzw. programmreaktionen ist denkbar. Ein interessanter repräsentant dieser klasse von testfallerzeugung ist das werkzeug *fuzz* von Miller u. a. (1990). Hierbei handelt es sich um ein programm, welches entsprechend vorgegebener konfigurationsdateien und möglicherweise unter zuhilfenahme eines sog. pseudoterminals eingabedaten, in diesem fall für UNIX-dienstprogramme, erzeugt. Ein überraschend grosser anteil von UNIX-dienstprogrammen konnte hiermit zum absturz gebracht bzw. in endlosschleifen geschickt werden; siehe die tabelle im zitierten artikel.

Wenn man die datenbezogene testmethode etwas verfeinert, so teilt man die eingaben in mengen diskreter werte, zusammenhängende wertebereiche

und daten mit expliziten beziehungen zu anderen daten ein; siehe hierzu das buch von Myers (1979). Die mengen diskreter werte teilt man dann in äquivalenzklassen ein, wobei daten aus der gleichen äquivalenzklasse eine gleiche wirkung im programm erzeugen sollen. Das testen mit repräsentativen werten aus einer äquivalenzklasse nennt Myers die *repräsentative wertanalyse*. Zusammenhängende wertebereiche testet man mit hilfe ihrer grenzwerte, wobei man sich in der regel auf den unteren grenzwert, den um 1 verminderten unteren grenzwert, den oberen grenzwert, den um 1 vermehrten oberen grenzwert und sicherheitshalber noch einen mittelwert beschränkt[28]. Dies nennt Myers die *grenzwertanalyse*. Bei datenfeldern mit definierten relationen zu anderen datenfeldern versucht man invariante relationen aufzuspüren und die erhaltung dieser invarianten durch das programm durch gezielt ausgewählte testfälle zu überprüfen. Unter umständen kann ein gutes datenlexikon zur auswahl von testfällen hier entscheidend beitragen.

5.4.3 Funktionsbezogenes testen

Beim funktionsbezogenen testen steht die spezifikation der programmfunktionen im vordergrund. Entsprechend dieser spezifikation werden testfälle ausgewählt. Die soeben angesprochenen grenzwerte von Myers können dabei sinnvollerweise herangezogen werden.

Funktionsbezogenes testen ist im grunde die sinnvollste art des softwaretests, da hierbei wirklich der kontakt zur spezifikation hergestellt wird. Die *unit tests*, wie sie durch werkzeuge à la JUnit unterstützt werden, gehören genau zu dieser kategorie.

5.4.4 Back-to-back-testen

Beim sog. *back-to-back*-testen werden mehrere versionen des gleichen programms bezüglich ihrer funktionalität miteinander verglichen. Eine wichtige variante hiervon ist der sog. *regressionstest*, den alle seriösen softwarehersteller verwenden. In einer datenbank werden alle jemals im leben des programms aufgefundenen fehler registriert. Nach einem änderungszyklus im programm bzw. vor der freigabe einer neuen version (*release*) werden dann alle bekannten testfälle aus der regressionsdatenbank erneut durchgespielt. Dies kann ein zeitraubender prozess sein, s. fussnote auf s. 108.

[28]Das klingt gerade so, als ob man hier ein „stetigkeitsargument“ verwenden würde, obwohl programme als digitale objekte ja maximal unstetig sind. Trotzdem werden mit dieser methode viele fehler gefunden.

5.4.5 Die rolle von zusicherungen

Voas (1997) bemerkt, dass der zweck eines dynamischen softwaretests ist, inkorrekte programmausgaben zu provozieren, da diese auf fehler im programm hinweisen. Damit dies überhaupt funktionieren kann, müssen folgende bedingungen erfüllt werden:

1. Eine eingabe muss dazu führen, dass fehlerhafter code ausgeführt wird.
2. Wenn fehlerhafter code ausgeführt wurde, muss der nachfolgende datenzustand des programms fehlerhaft werden.
3. Ein datenzustandsfehler muss sich in die programmausgabe fortpflanzen, m. a .w. wir müssen eine fehlerhafte ausgabe erhalten.

Die erste bedingung hiervon lässt sich nur durch eine geschickte auswahl von testfällen erreichen. In manchen programmiersprachen gibt es *zusicherungen* („*assertions*"), die eine Boolesche aussage beinhalten und zur erzeugung einer ausnahmebedingung („*exception*") führen, wenn zur laufzeit diese zusicherungen ungültig sind. Wir haben dies hier am beispiel der JML kennengelernt. Mit diesem mechanismus kann eine entsprechende zusicherung über den korrekten datenzustand dafür sorgen, dass auch bedingungen 2 und 3 erfüllt werden.

5.5 Statische analyse

Die bis jetzt angesprochenen dynamischen analysemethoden sind per definitionem nur für fertige, fehlerfrei übersetzte programme zu gebrauchen. Wir haben aber bereits zuvor erkannt, dass wir auch bereits auf der stufe der analyse und des entwurfs qualitätssicherungsmassnahmen einsetzen müssen. Diese massnahmen können per definitionem nur statisch sein, d. h. sie müssen ohne programmlauf auskommen. Bedenkt man, dass analyse- und entwurfsfehler viel teurer zu beseitigen sind als reine codierfehler, so kommt diesem bereich sogar noch grössere bedeutung zu als den dynamischen analysemethoden.

5.5.1 Vollständigkeit und konsistenz

Eine erste möglichkeit der statischen analyse ist es, den zieltext auf vollständigkeit und konsistenz zu prüfen. Dies ist auch für Dokumente aus der Problemanalyse und dem Systementwurf bereits möglich.

Auf der programmstufe ist die situation recht unterschiedlich: Es gibt sog. sichere programmiersprachen, die relativ restriktive bedingungen, vor allem

im bereich der kontextsensitiven syntax, haben (etwa Pascal, Modula-2, Oberon, Ada, Eiffel). Hier ist bereits eine erfolgreiche compilation ein gutes indiz für das fehlen von unvollständigkeiten und inkonsistenzen. Andere programmiersprachen, wie z. B. C++ oder vor allem C sind wesentlich permissiver. Hier erlaubt der compiler dem programmierer eine ganze reihe freiheiten, die ausgesprochen fehlerträchtig sind. Ein separates werkzeug namens `lint` führt auf wunsch erweiterte analysen am programmtext durch; sehr häufig wird von managern verlangt, dass abgelieferte programme ohne warnmeldungen durch `lint` bearbeitet wurden. Zur leistung von `lint` zitiere ich aus der onlinehilfe von AIX 4.1:

> *Das lint-Kommando prüft Quellcode der Sprachen C und C++ auf Codier- und Syntaxfehler und auf ineffizienten oder nicht portablen Code. Sie können dieses Programm verwenden, um Inkompatibilitäten zwischen Quellprogrammen und Bibliotheken aufzudecken; Typüberprüfungsregeln strenger zu erzwingen als der Compiler; mögliche Probleme mit Variablen und Funktionen aufzudecken; Probleme mit der Ablaufsteuerung aufzudecken; legale Konstruktionen aufzudecken, die Fehler produzieren oder ineffizient sein können; unbenutzte Variablen- und Funktionsdeklarationen aufzudecken und möglicherweise nicht portablen Code aufzudecken. Die Verwendung von Funktionen wird über Dateigrenzen hinweg geprüft, um Funktionen zu finden, welche in einigen Situationen Resultate abliefern, aber in anderen nicht, sowie Funktionen, die mit variierenden Anzahlen oder Typen von Argumenten aufgerufen werden und Funktionen, deren Werte nicht benutzt werden oder deren Werte zwar benutzt, aber nicht zurückgegeben werden.*

Auch programmiersprachen der zuvor angesprochenen klasse sicherer programmiersprachen erlauben es jedoch dem programmierer, eine reihe von fehlern zu machen; selbstverständlich werden ungenutzte variablen und prozedurdeklarationen vom compiler ebenso akzeptiert wie nicht erreichbarer code. Zahlreiche möglichkeiten für missverständnisse des programmierers liegen auch in dem von Algol 60 übernommenen blockkonzept. Hier ist es möglich, eine variable mit gleichem namen auf unterschiedlichen schachtelungsstufen zu definieren. Damit sind automatisch die folgenden beiden fehlermöglichkeiten vorgezeichnet, die sich nur aufgrund intimer semantischer kenntnis des programms untersuchen lassen und jeder vollständigen behandlung durch programmierwerkzeuge entzogen sind:

1. Eine funktion verändert eine variable, die in einem äusseren sichtbarkeitsbereich deklariert wurde (nicht-lokale variable). Selbstverständlich kann dies genau der gewünschte effekt sein, es ist jedoch auch möglich, dass der programmierer schlicht vergessen hat, diese variable in dem eingeschachtelten sichtbarkeitsbereich neu zu deklarieren.

2. Ein eingeschachtelter sichtbarkeitsbereich deklariert eine variable eines äusseren sichtbarkeitsbereichs neu. Dann kann der programmierer jedoch

an jeder verwendungsstelle, vor allem bei grösseren programmen, die beiden deklarationen miteinander verwechseln,

Interessante ansätze, die über die leistung von `lint` hinausgehen, sind z. B. das „Secure Programming Lint/SPecifications Lint“-projekt der University of Virginia (`www.splint.org`). Die arbeit von Louridas (2006) skizziert einige interessante werkzeuge für die statische analyse von Java-programmen.

In einem sehr lesenswerten artikel diskutieren Bessey u. a. (2010) ihr werkzeug *Coverity* und die schwierigkeiten, die auftreten, wenn ein im akademischen umfeld gut funktionierender ansatz in die „richtige welt“ entlassen wird. Einerseits ist die schiere menge aufgefundener fehler oft schockierend und wird von programmierern gern als „falsch-positiv“ charakterisiert, andererseits ist die abweichung der compiler von der sprachdefinition bemerkenswert. Programmierer betrachten code, den „ihr“ compiler ohne fehlermeldungen akzeptiert, als syntaktisch korrekt, egal, wie falsch er laut sprachdefinition sein mag:

> The C language does not exist; neither does Java, C++, and C#. While a language may exist as an abstract idea, and even have a pile of paper (a standard) purporting to define it, a standard is not a compiler. What language do people write code in? The character strings accepted by their compiler. Further, they equate compilation with certification. A file their compiler does not reject has been certified as „C code“ no matter how blatantly illegal its contents may be to a language scholar.

Die arbeit von Drake (1996) zeigt eine reihe interessanter masse auf, die durch werkzeuge mittels statischer analyse von programmen gewonnen werden können, und die erfahrungsgemäss grossen einfluss auf die programmqualität haben. Abb. 35 zeigt eine solche auswertung für ein programm in form eines sogenannten *Kiviat-Diagramms*. Diese form von diagrammen eignet sich besonders für den fall, wo eine vielzahl von metriken bezüglich ihrer einhaltung bestimmter unter- und obergrenzen dargestellt werden soll. Es gibt hier einen inneren kreis, beschriftet mit dem jeweiligen minimum für die einzelnen metriken, und einen äusseren kreis mit den maxima; die messwerte werden alle entsprechend skaliert und durch eine linie verbunden. Das bild zeigt, dass einige messwerte – über deren bedeutung wir hier nicht diskutieren wollen – stark ausserhalb der zulässigen grenzen liegen. Drake nennt die hier entstandene figur den „fliegenden albatros“ und sagt dazu, dass es ein zeichen für ein besonders schlechtes programm ist. Die verbesserung des programms aufgrund der so gewonnenen erkenntnisse zeigt dann ein bild wie in abb. 36.

Ähnlich wertvolle erkenntnisse lassen sich durch analyse der schleifenstruktur gewinnen; abb. 37 zeigt ein zu kompliziertes programm und abb. 38 das überarbeitete programm.

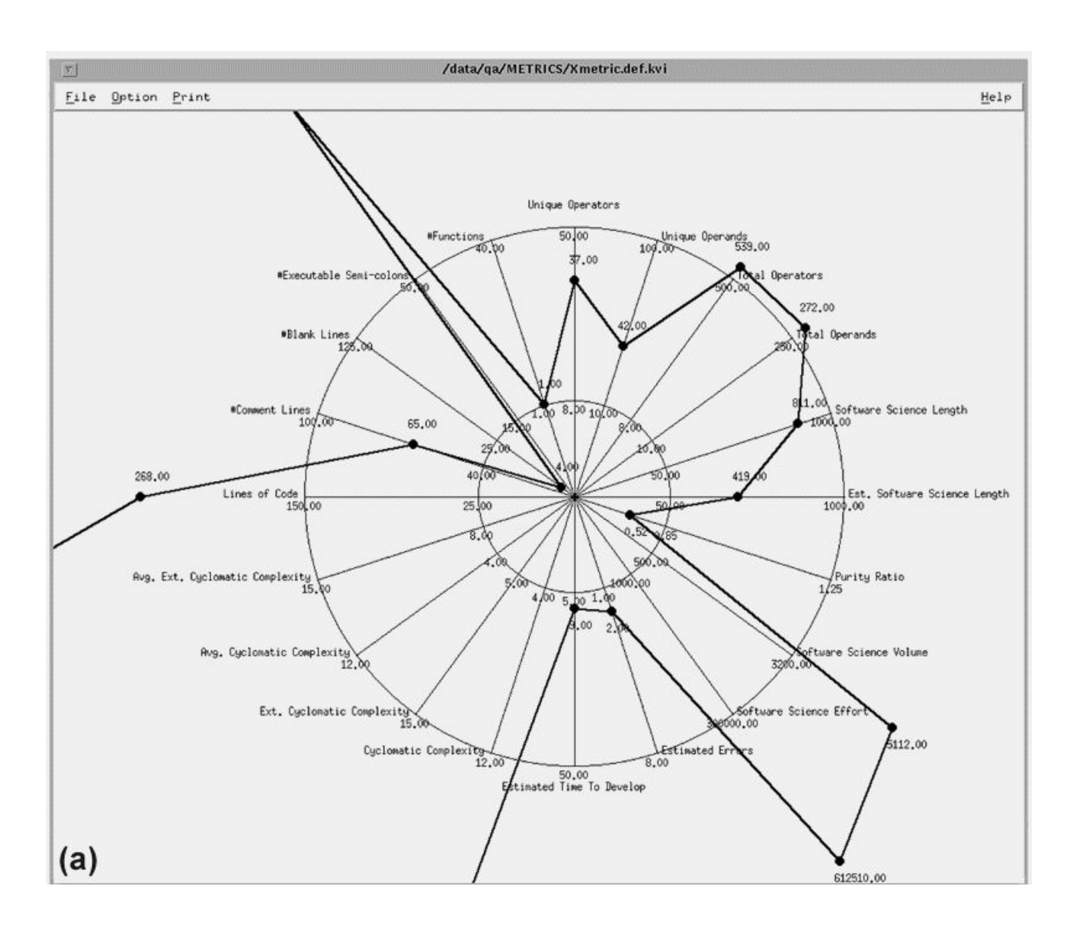


Abbildung 35: Kiviat-Diagramm eines schlechten programms

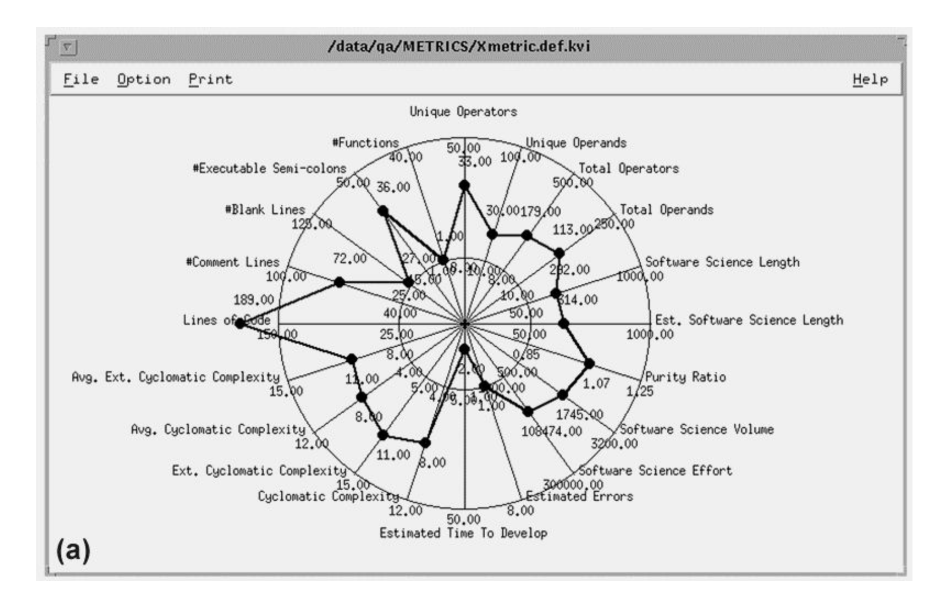


Abbildung 36: Kiviat-Diagramm eines guten programms

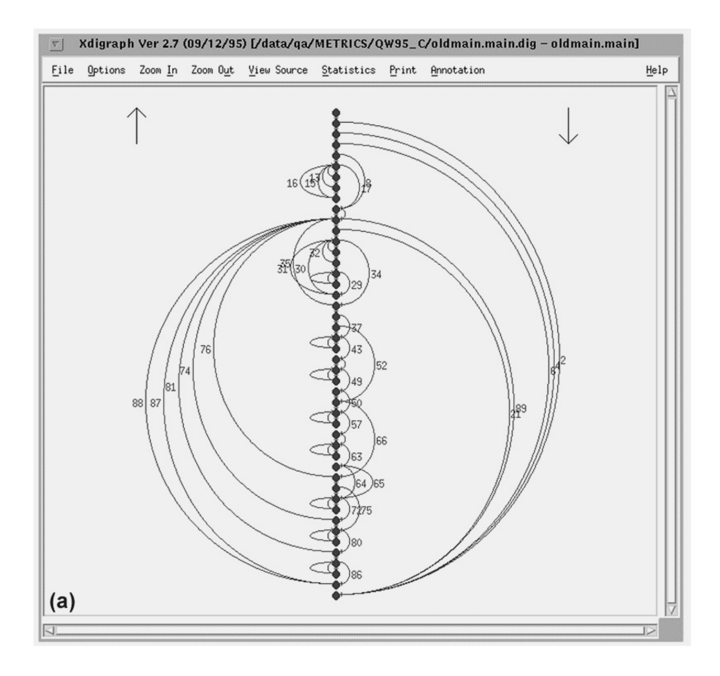


Abbildung 37: Schleifenstruktur eines schlechten programms

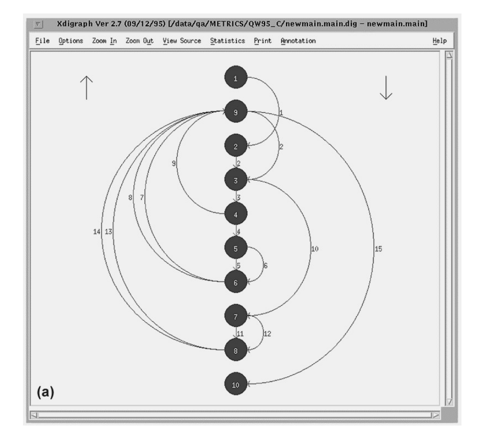


Abbildung 38: Verbesserte schleifenstruktur

Nebenbei sei bemerkt, dass solche werkzeuge zur statischen analyse von programmen ähnliche methoden verwenden wie die, die man für den compilerbau braucht.

5.5.2 Inspektionen

Eine andere methode der software-qualitätssicherung ist es, programm-, entwurfs- oder analysedokumente manuell auf die einhaltung bestimmter kriterien oder auf vollständigkeit, konsistenz, plausibilität etc. zu prüfen. Dies geschieht im rahmen von formell veranstalteten besprechungen der dokumente im entwurfsteam. Hierfür haben sich begriffe wie inspektion, *„code review"*, *„structured walkthrough"*, *„formal technical review"* und ähnliche etabliert. Rothman (1999) erwähnt neben anderen bekannten vorteilen dieser methode auch pädagogische bzw. organisatorische:

> *Wir zeigten der ganzen firma, wie man gute programme schreibt, und wir halfen dem ganzen team, das produkt kennenzulernen.*

Die strengste interpretation des begriffs „inspektion" kommt aus dem hause IBM und geht auf Fagan (1976) zurück. Die genaue ausprägung inspektionsartiger prozesse ist jedoch nicht so wichtig wie die grundtatsache, dass diese prozesse mit genau definierten rollen und präzise eingehaltenen spielregeln ablaufen; je nach projekt oder firma mögen diese dann unterschiedlich angelegt sein.

Fagans inspektionen gehen nach einem festgeschriebenen *drehbuch* vor. Ihr ziel ist es, zwei dokumente in bezug auf konsistenz, korrektheit und vollständigkeit zu überprüfen (Schnurer 1988). Das zu inspizierende dokument, auch I-dokument genannt („I" wie „Implementation") wird hier verglichen mit einem S-dokument („S" wie „Spezifikation"). Normalerweise stammt das S-dokument aus einem vorherigen bereits abgeschlossenen entwicklungsschritt. Ziel des inspektionsprozesses ist es, eine möglichst grosse anzahl von fehlern zum frühestmöglichen zeitpunkt aufzudecken. In der praxis werden hierbei aufgrund statistischer erfahrungsdaten konkrete vorgaben für die anzahl der fehler gemacht, die vermutlich im produkt vorhanden sind und deshalb im inspektionsprozess zu finden sind. Auf einer meta-ebene der software-qualitätssicherung geht es dann auch noch darum, den software-entwicklungs- und herstellungsprozess sowie den inspektionsprozess selbst stetig zu verbessern. Zu einer inspektion gehören mindestens drei bestandteile:

Eingangskriterien (*entry criteria*) Wenn nicht ganz bestimmte, je nach art des zu prüfenden dokuments unterschiedliche vorbedingungen erfüllt sind, kann der inspektionsprozess nicht beginnen.

Sitzung Wenn die vorbedingungen erfüllt sind, kann eine inspektionssitzung als eine art rollenspiel anberaumt werden. Zu dieser sitzung erscheinen 4 personengruppen:

- Softwareentwickler
- Softwaretester
- Moderator
- Protokollant

Während der inspektionssitzung wird das zu inspizierende dokument im wesentlichen zeile für zeile durchgesprochen.

Ausgangskriterien (*exit criteria*) Diese kriterien legen fest, wann ein einzelner inspektionsprozess als beendet gelten kann.

Entwicklungsschritt	Min	Med	Max
Funkt.-Spez.-Inspektion	2	3	5
Grobentwurf-Inspektion	10	12	16
Feinentwurf-Inspektion	17	20	22
Code-Inspektion	20	28	32
Modul-Test	12	16	17
Funktionen-Test	10	12	14
Komponenten-Test	8	7	14
System-Test	0,5	2	7

Tabelle 7: Prozent der aufgefundenen fehler

Die bei der inspektion anfallenden daten werden in einer projekt-datenbank gespeichert und analysiert. Hiermit werden einerseits hinweise auf schwachstellen im entwicklungsprozess deutlich, andererseits unterwirft sich damit aber selbstverständlich auch der inspektionsprozess selbst einer bewertung. In tab. 7 finden wir daten, wieviel prozent der insgesamt bei IBM vor auslieferung eines programms gefundenen fehler auf einzelne qualitätssicherungsmassnahmen zurückzuführen sind. Durch summation der werte erkennen wir, dass 63%, also fast zwei drittel der fehler durch inspektionen auf den verschiedenen niveaus gefunden werden.

Schnurer berichtet, dass in einem compiler-projekt bei der inspektion jeweils ca. 2,2 mannstunden zur aufdeckung eines fehlers notwendig waren, während

beim testen immerhin 24,4 stunden pro fehler investiert werden mussten. Damit kommen wir auf einen produktivitätsvorteil von 12:1 zugunsten der inspektionen. Beim betriebssystem DOS VSE wurde dagegen ein verhältnis von 3:1 gemessen.

Die konstruktion des prozesses als *wettbewerb* zwischen *software-entwicklern*, deren interesse es sein muss, möglichst fehlerfreie software abzuliefern und *software-testern*, deren aufgabe es ist, möglichst viele fehler aufzudecken, sorgt für eine atmosphäre, in der eine möglichst grosse software-qualität erreicht wird.

Aufgabe des moderators[29] ist es hierbei, einerseits aggressionen zu verhindern, die aus der wettbewerbssituation entstehen könnten, andererseits aber auch dafür zu sorgen, dass die beteiligten sich ihre aufgabe nicht zu leicht machen. Dazu gehört die möglichkeit, inspektionen bei vorliegen bestimmter kriterien abzubrechen und re-inspektionen anzusetzen. Dieser fall tritt beispielshalber ein, wenn während einer sitzung unerwartet viele, aber auch wenn unerwartet wenige fehler gefunden wurden. Solche re-inspektionen sind auf allen seiten gefürchtet, weil sie die gefahr in sich bergen, dass der gesamtprozess verzögert wird.

In etwas grösserem detaillierungsgrad besteht der inspektionsprozess aus den folgenden schritten:

Planung Der projektleiter benennt moderator und inspektoren. Der moderator prüft, ob das inspektionsmaterial die eingangskriterien erfüllt, setzt zeit und ort für die inspektion fest und verteilt ca. 2 wochen vorher das zu inspizierende material.

Einführung Falls nötig, wird in einer gesonderten sitzung die problematik des zu inspizierenden materials dargelegt.

Vorbereitung Die teilnehmer erarbeiten sich anhand der verteilten unterlagen ein verständnis von dem zu inspizierenden teilprodukt und notieren sich listen von fragen über das produkt.

Inspektion Dies ist die eigentliche sitzung, in der das produkt zeile für zeile durchdiskutiert wird und alle fragen aus der vorbereitung der inspektoren besprochen werden. Aus dieser sitzung ergibt sich eine agenda-liste für die am produkt durchzuführenden änderungen.

Korrektur Innerhalb einer jeweils individuell festgelegten zeitspanne korrigiert der autor die während der inspektionssitzung angesprochenen probleme.

[29] lat. *moderare* = mässigen

Nacharbeit Der moderator prüft, ob alle fehler behandelt wurden bzw. behoben sind und ob gegebenenfalls eine erneute inspektion des verbesserten produkts notwendig ist.

Eine wichtige, und wie Schnurer berichtet, manchmal umstrittene aufgabe des moderators ist das klassifizieren von fehlern. Fehler werden klassifiziert

nach typ: Ist das textstück in bezug auf die spezifizierte aufgabe unvollständig, unkorrekt oder überflüssig?

nach schwere: Hier wird unterschieden zwischen *formalen fehlern*, d.h. verstössen gegen programmiernormen, unzureichende oder missverständliche kommentare und ähnliche leichtere vergehen; *funktionalen fehlern*, d.h. verstössen gegen die spezifikation des S-dokuments und *offenen fragen*. Dies sind in gewissem sinne die härtesten punkte, da hier zunächst uneinigkeit besteht, ob ein fehler vorliegt oder nicht. In der inspektionssitzung mögen die meinungen hierüber auseinandergehen. Eine protokollierung eines problempunkts als offene frage führt dazu, dass autor und einige inspektoren im anschluss an die inspektionssitzung dieser frage gesondert nachgehen und hierüber bericht erstatten müssen.

nach kategorien: z. b. datendefinition, registerbenutzung, logik, wartbarkeit, entwurffehler, normen, testabdeckung

nach fehlerquelle: welcher entwicklungsschritt hat diesen fehler verursacht.

Phase	Vorbereitung	Inspektion
Grobentwurf	200 LOC	220 LOC
Feinentwurf	100 LOC	100 LOC
Code	125 LOC	90 LOC

Tabelle 8: Inspektionsleistung pro stunde

IBM empfiehlt unterschiedliche anzahlen von personen für die verschiedenen stufen von inspektionen. Als grundsätzliche regel gilt, dass kleinere teams arbeitsfähiger sind als grössere. So wird etwa für code-inspektionen ein kreis von ca. 4 personen vorgeschlagen, für entwurfsspezifikationen können es auch 5–6 sein. Eine inspektion mit mehr als 8 teilnehmern ist erfahrungsgemäss sehr schwer zu moderieren. Auch gibt es präzise vorgaben darüber, wie schnell ein produkt in vorbereitungsphase und inspektionssitzung bearbeitet werden kann. Diese sind in tab. 8 enthalten.

Ein studentisches projekt an der uni Ulm (Ernst u. a. 1997) hat die erfahrung bestätigt, dass inspektionen mehr fehler finden als tests. Inspektionen lieferten

ausserdem mehr vorschläge zur codeverbesserung. An vier beispielen eingebetteter systeme (jeweils weniger als ein halbes kLOC) zeigte sich, dass beim testen durchschnittlich 0,22 meldungen pro mannstunde erzeugt wurden, bei der inspektion dagegen 0,59 meldungen pro mannstunde, d. h. man kam hier auch ungefähr auf den von Schnurer für DOS VSE berichteten produktivitätsvorteil von 3:1. Untersucht wurde dabei auch die wichtigkeit der inspektionssitzung; die studenten hatten hier den eindruck, dass diese im prinzip entbehrlich sei: die eigentliche arbeit läge in der vorbereitung der inspektoren, und der rest könne dann auch schriftlich erledigt werden. Das experiment zeigte dann aber, dass die vorbereitung auf die inspektion nur ein teil der angelegenheit ist; die sitzung mit der diskussion ist sehr wichtig. 34% der meldungen wurden erst während der sitzung erzeugt; sie waren auf den vorbereitungslisten nicht enthalten.

Dunsmore u. a. (2003) listen die besonderheiten auf, mit denen man im inspektionsprozess von OO systemen zu kämpfen hat, u.a. den „delokalisierungseffekt“: Da OO systeme mit zahlreichen verweisen (nach der art der typischen deutschen behörde) arbeiten und ausserdem noch vererbungshierarchien ins spiel kommen, ist die information, die man zum verstehen weniger codezeilen braucht, häufig über weite teile des systems verstreut.

TODO: Neville-Neil (2009) gibt praktische hinweise auf die durchführung von code reviews und erwähnt das Google tool Rietveld. (`http://code.google.com/p/rietveld/`)

5.6 Effizienz der fehlerbeseitigung

Tätigkeit	*Effizienz*
Informelle entwurfsinspektion	25% – 40%
Formelle entwurfsinspektion	45% – 65%
Informelle programminspektion	20% – 35%
Formelle programminspektion	45% – 70%
Modultest	15% – 50%
Test neuer funktionen	20% – 35%
Regressionstest	15% – 30%
Integrationstest	25% – 40%
Systemtest	25% – 55%
Kleiner betatest (bis 10 kunden)	25% – 40%
Grosser betatest (über 1000 kunden)	60% – 85%

Tabelle 9: Wirkung einzelner QS-massnahmen

Selbstverständlich sind nicht alle methoden zur fehlersuche gleichermassen effizient. Die effizienz dieser methoden wird beispielshalber von Jones (1996, 1997) verglichen, wobei er effizienz definiert als den quotienten zwischen der anzahl der im rahmen der qualitätssicherung bei der software-entwicklung gefundenen fehler zu der gesamtzahl der fehler nach einer gewissen zeit des betriebs des fertigen produkts. Eine effizienz von 90% bedeutet deshalb, dass der qualitätssicherungsprozess 90% der fehler auffindet, die sonst beim kunden auftreten würden. Damit sind fehler, die auch beim kunden nicht auftreten, automatisch von der betrachtung ausgeschlossen.

Auf der basis empirischer daten von AT&T, BellCore, Bell Sygma, Bell Northern Research, Hewlett Packard, IBM, Microsoft, Motorola, Siemens Nixdorf und einigen hundert kleineren firmen wird etwa in tab. 9 zusammengefasst, dass zwischen 25% und 40% der insgesamt aufgetretenen fehler durch eine informelle inspektion des entwurfs gefunden werden können. Besser ist das ergebnis bei formalen inspektionen des programm-codes (45% bis 70%). Jones bemerkt ausserdem, dass jede kombination von verfahren, die nicht mindestens 85% effizienz in der fehlerbeseitigung bringt, unakzeptabel ist, da solche programme vom benutzer als unzuverlässig betrachtet werden. Leider sind diese 85% im augenblick der durchschnitt der effizienz in der US-amerikanischen software-industrie. Nur die „klassenbesten“ ermitteln wirklich selbst die effizienz der fehlerbeseitigung und diese kommen alle über 96%. Führende firmen (hier nennt er AT&T, IBM, Motorola, Raytheon und Hewlett Packard) erreichen jedoch in ihren besten projekten mehr als 99% fehlerbeseitigungseffizienz.

In diesem zusammenhang ist es dann interessant, die kombination bestimmter verfahren zu untersuchen. Die ergebnisse dieser untersuchung sind in tab. 10 zusammengetragen. In dieser tabelle ist die effizienz jeder denkbaren kombination von vier techniken zusammengetragen, nämlich

DI (*Design Inspection*) Formelle inspektion des entwurfs

CI (*Code Inspection*) Formelle inspektion des programm-codes

QA (*Quality Assurance*) Formelle qualitätssicherungsverfahren

FT (*Formal Testing*) Formelles testen der software

Für jede solche kombination sind die besten und schlechtesten angetroffenen werte aufgelistet sowie der median der werte; die zeilen sind dann nach median aufsteigend sortiert. Wie nicht anders zu erwarten, nimmt die effizienz zu, je mehr dieser techniken gleichzeitig verwendet werden. Es fällt jedoch

DI	*CI*	*QA*	*FT*	*Min*	*Med*	*Max*
				30%	40%	50%
		X		32%	45%	55%
			X	37%	53%	60%
	X			43%	57%	66%
X				45%	60%	68%
		X	X	50%	65%	75%
	X	X		53%	68%	78%
	X		X	55%	70%	80%
X		X		60%	75%	85%
X			X	65%	80%	87%
X	X			70%	85%	90%
	X	X	X	75%	87%	93%
X		X	X	77%	90%	95%
X	X	X		83%	95%	97%
X	X		X	85%	97%	99%
X	X	X	X	95%	99%	99.99%

Tabelle 10: Kombinierte wirkung von QS-strategien

auf, dass die besseren zahlen praktisch nur mit hilfe formeller entwurfsinspektionen zu erzielen sind; kombiniert man diese noch mit formellen inspektionen des programmcodes, so erreicht man bereits die genannten 85% effizienz. Nimmt man jetzt noch formelles testen der software hinzu, so kommt man bereits auf 99%, was als ausgezeichnet zu betrachten ist.

5.7 Value-driven testing (2)

Um die wirtschaftlichkeit von tests zu belegen, entwickeln Sneed und Jungmayr (2011) für den testaufwand die formel

$$A = AF \cdot \left[\left(\frac{TF \cdot TU}{TP + TP \cdot (1 - TA)} \right)^{TE} \cdot \frac{MT}{TB} \right]$$

Dabei bedeutet

A	Testaufwand
AF	Korrekturfaktor (Standardwert: 1)
TF	Anzahl der Testfälle
TU	Testüberdeckung
TP	manuelle Testproduktivität
TA	Testautomatisierungsgrad
TE	Skalierungsexponent (Standardwert: 1,03)
MT	mittlere Testbarkeit
TB	Testbarkeitsmetrik (Standardwert: 0,45)

AF, TE, MT und TB sind erfahrungswerte, die über die zeit hin durch gemessene werte verbessert werden müssen. TF ergibt sich aus der anzahl der anweisungen im zu testenden programm; Sneed und Jungmayr (2011) behaupten, dass ca. ein testfall pro 20 zeilen eines modernen objektorientierten programms notwendig ist. TU gibt an, welcher prozentsatz dieser theoretisch nötigen testfälle wirklich durchgeführt werden soll.

Die berechnung wird an einem beispiel vorgeführt:

- 100.000 anweisungen, ergibt 5000 testfälle
- testüberdeckung TU = 67 %
- testproduktivität TP = 4 testfälle pro tag
- automatisierungsgrad TA = 40 %

Angenommen wird ausserdem eine fehlerrate von 6 fehlern pro 1000 anweisungen, das ergibt insgesamt 600 fehler, die zunächst im produkt verbleiben. Die berechnung des aufwands ergibt also

$$A = 1 \cdot \left[\left(\frac{5000 \cdot 0,67}{4 + (4 \cdot (1 - 0,4))} \right)^{1,03} \cdot \frac{0,5}{0,45} \right] = \left(\frac{3350}{6,4} \right)^{1,03} \cdot 1,11 = 700$$

Es sind also für diese vorgaben 700 personentage für den test vorzusehen. Nun kann die eigentliche berechnung des test-ROI beginnen. Dazu treffen wir die folgenden zusätzlichen annahmen:

- Bedienung des systems durch 100 sachbearbeiter, die 40 EUR je stunde verdienen,
- schwere systemausfälle legen das system für eine stunde lahm, leichte systemausfälle für eine halbe stunde, 10 % der fehler führen zu einem schweren systemausfall,
- falsche ergebnisse treten bei 50 % der fehler auf und kosten
  - im schlimmsten fall jeden sachbearbeiter eine halbe stunde,
  - im günstigsten fall nur 10 % der sachbearbeiter eine halbe stunde
- entwickler kosten 800 EUR pro tag,

- tester kosten 720 EUR pro tag.

Würden die 600 fehler im produkt verbleiben, so gäbe es also 60 ausfälle und 300 falsche ergebnisse; die kosten beim kunden wären im schlimmsten fall

$$60 \cdot 100 \cdot 40 \text{ EUR} + 300 \cdot 50 \cdot 40 \text{ EUR} = 840.000 \text{ EUR}$$

und im günstigsten fall

$$60 \cdot 50 \cdot 40 \text{ EUR} + 300 \cdot 10 \cdot 40 \text{ EUR} = 240.000 \text{ EUR},$$

im Mittel also

$$\frac{840.000 + 240.000}{2} \text{ EUR} = 540.000 \text{ EUR}$$

Dazu kommen die kosten für die fehlerbeseitigung. Erfahrungsgemäss teilen sich die 600 fehler wie folgt auf:

- 240 anforderungsfehler (= 40 %), beseitigung kostet 2,5 entwicklertage,
- 180 entwurfsfehler (= 30 %), beseitigung kostet 1 entwicklertag,
- 180 codierfehler (= 30 %), beseitigung kostet $\frac{1}{2}$ entwicklertag.

Bei der angenommenen testüberdeckung von $\frac{2}{3}$ würden die 600 fehler auf 200 reduziert, es müssten also 400 fehler (160 anforderungsfehler, 120 entwurfsfehler, 120 codierfehler) nicht beseitigt werden. Das ergibt ein einsparungspotential von

$$160 \cdot 2,5 + 120 + 120 \cdot 0,5 = 580$$

entwicklertagen, entsprechend einem betrag von 464.000 EUR. Rechnet man dazu die kosten beim kunden (540.000 EUR), so ergibt sich insgesamt ein vorteil von 1.004.000 EUR durch die testmassnahmen. Dieser muss verglichen werden mit den kosten für das testen: die 700 personentage kosten 504.000, also ist der ROI für das testen (definiert als die differenz zwischen nutzen und kosten, geteilt durch die kosten):

$$\frac{1.004.000 - 504.000}{504.000} = 0,99$$

Mit anderen Worten: Jede 100 EUR, die ins testen investiert werden, sparen 99 EUR für die später nicht auftretenden fehler.

5.8 Verifizieren

Die idee, programme formal zu verifizieren, geht auf einen alten artikel von Tony Hoare (1969) zurück, der grossen einfluss auf die entwicklung von informatikmethoden gehabt hat. Vierzig jahre später zieht Hoare (2009) ein resumé und bemerkt dabei, dass seine damalige ansicht, verifizieren sei eine (bessere) alternative zum testen, falsch gewesen sei:

> *I did not realize that the success of tests is that they test the programmer, not the program.*

Jeder fehlgeschlagene testfall macht den programmierer darauf aufmerksam, dass er nicht sorgfältig genug gearbeitet hat und trägt damit zur verbesserung des programmierers ebenso bei wie zur verbesserung des programms:

> *The experience, judgment, and intuition of programmers who have survived the rigors of testing are what make programs of the present day useful, efficient, and (nearly) correct. Formal methods for achieving correctness must support the intuitive judgment of programmers, not replace it.*

Verifizieren ist also keine alternative zum testen, sondern vielmehr ein anderer aspekt des problems:

> *My basic mistake was to set up proof in opposition to testing, where in fact both of them are valuable and mutally supportive ways of accumulating evidence of the correctness and serviceability of programs.*

Er weist anschliessend darauf hin, wie seine *verification conditions* später in von ihm nicht vorhergesehener weise als *assertions* (abschn. 5.4.5) eingang in den alltag des programmierers gefunden haben; auch der *design by contract* (abschn. 2.4) basiert letztlich auf den alten verifikationsideen.

Voraussetzung für das verifizieren ist natürlich eine formale spezifikation für das programm. Am anfang dieses kapitels wurde bereits diskutiert, wie schwer es sein kann, eine solche spezifikation herzustellen. Hat man sie jedoch, so kann man mit mitteln der mathematischen logik beweisen, dass das programm dieser spezifikation entspricht. Im einfachsten fall ergibt sich nur eine ja/nein-antwort; in wirklichkeit wollen wir aber nicht nur erfahren, dass das programm nicht korrekt ist, sondern wir wollen präzise wissen, *wo* der fehler steckt.

An der universität Ulm wurden mit Hilfe des KIV (Karlsruhe Interactive Verifier) einige kritische softwaremodule verifiziert (z. b. zugangskontrolle für ein kernkraftwerk, crash control für airbags) und dabei mehrere hundert fehler aufgedeckt. Es hat sich dabei gezeigt, dass ein experte im jahr zwischen 1000 und 2000 zeilen programm verifizieren kann. Das sind immerhin 50 bis 100 EUR pro zeile nur für die verifikation; dafür ist die software dann per definitionem fehlerfrei. Die verifikation der gesamten Space-Shuttle-software würde allerdings nach diesen massstäben einen experten zwischen 1500 und 3000 jahren beschäftigen.

Kontrollfragen

1. Zählen sie die nach Boehm und Basili 10 wichtigsten erkenntnisse zur fehlerreduktion in programmen auf!
2. Was versteht man unter einer Pareto-verteilung? Nennen sie drei beispiele, wo solche verteilungen in der diskussion um softwarequalität vorkommen!
3. Welche fehlerdichten gibt es in kommerziell vertriebenen programmen?
4. Wieso ist software mit wenig fehlern nicht teurer als software mit vielen fehlern?
5. Was spricht gegen das sog. *error seeding*?
6. Ordnen sie die testmethoden ablaufbezogenes, datenbezogenes und funktionsbezogenes testen den kategorien *black box test* und *white box test* zu.
7. Welche arten von fehlern lassen sich mit dem ablaufbezogenen testen finden? Welche lassen sich mit dieser methode bestimmt nicht finden?
8. Beschreiben sie den prozess der programminspektion nach Fagan!
9. Welche inspektionsleistungen pro stunde werden empfohlen?
10. Wie verhalten sich statische und dynamische analysemethoden zueinander in bezug auf den prozentsatz der aufgefundenen fehler?
11. Welche massnahmen kann man treffen, um eine fehlerbereinigungseffizienz von 86% zu erreichen? Wieso sind diese 85% so wichtig?

6 Vorgehensmodelle für die softwareentwicklung

6.1 Software-lebensläufe

The same reused software components that killed people when used to control the Therac-25 had no dangerous effects in the Therac-20. Safety does not even exist as a concept when looking only at a piece of software — it is a property that arises when the software is used within a particular overall system design. Individual components of a system cannot be evaluated for safety without knowing the context within which the components will be used. (Leveson 1995)

A NYC subway train on the Williamsburg bridge crashed into the rear end of another train on 5 June 1995. The motorman apparently ran through a red light. (See news service reports on 6 and 7 June 1995). The safety system did apply emergency brakes, as expected. However, the safety parameters and signal spacing were set in 1918, when trains were shorter, lighter, and slower, and the emergency brake system could not stop the train in time. (See The New York Times*, 16 Jun 1995, noted by Mark Stalzer). (Neumann 1995b)*

Bereits in den vergangenen abschnitten haben wir von *analysephasen*, *entwurfsphasen* und *implementierungsphasen* gesprochen. Damit wird ein gewisses *vorgehensmodell* für die softwareentwicklung angesprochen. Hier wollen wir uns jetzt mit solchen vorgehensmodellen näher auseinandersetzen. Dabei kommt man normalerweise auch auf die softwareentwicklung als nur einer teilaufgabe bei der systementwicklung zu sprechen. Die einleitenden zitate machen klar, dass wir die folgenden konzepte unterscheiden müssen:

Systeme bestehend aus *hard- und software* und

Software als solche. Wir beschäftigen uns zwar mit software engineering, aber man kann software nicht losgelöst betrachten.

Bei einer systementwicklung erfordern also die vielfältigen interaktionen zwischen software und hardware sorgfältige beobachtung. Wir diskutieren aber im übrigen nur über software.

Vorgehensmodelle sind etwas allgemeines: sie beschreiben einen plan, wie man die softwareentwicklung organisieren möchte. Daraus entwickeln sich dann individuelle *software-lebensläufe*, je nachdem wie einzelne projekte dann tatsächlich verlaufen. Häufig findet jedoch in der literatur (und auch in dieser vorlesung) eine leichte begriffsverwirrung statt, indem nämlich auch das vorgehensmodell selbst als software-lebenslauf („software life cycle“) bezeichnet wird.

Vorgehensmodelle für die softwareentwicklung gibt es in grosser zahl. Vielfach sind sie an die speziellen gepflogenheiten eines bestimmten teams oder

einer firma angepasst oder nehmen rücksicht auf die gegebenheiten bestimmter software-systeme zur unterstützung der software-entwicklung (sogenannte Software Engineering Environments). Ausserdem bestehen unterschiede darin, wie fein einzelne tätigkeiten dabei untergliedert sind und wieweit rückgriffe im entwicklungsprozess dargestellt werden. Das einfachste modell (das sich aber so nirgendwo findet) ist in abb. 39 enthalten. Es passt nicht nur auf

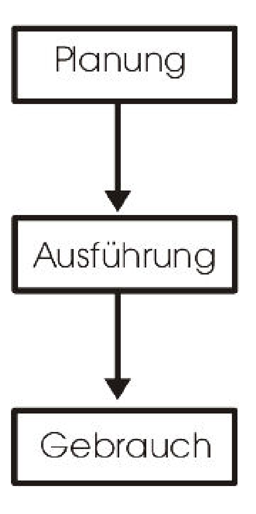


Abbildung 39: Trivialer software-lebenslauf

die software-entwicklung, sondern auf fast alle tätigkeiten, bei denen irgendetwas hergestellt wird, beispielsweise auch auf den bau eines wohnhauses. Man erkennt aber auch drei grundsätzliche mängel an diesem schema, die im prinzip von allen solchen vorgehensmodellen geteilt werden:

Abgrenzungsproblem Während das schema klar voneinander abgegrenzte phasen vortäuscht, kann in wirklichkeit der übergang fliessend sein. Speziell bei ganz grossen projekten können sich einzelne teile noch in der planung befinden, während andere schon in der ausführung oder gar schon im gebrauch sind.

Abfolgeproblem Während das schema vortäuscht, dass die einzelnen phasen streng zeitlich nacheinander ablaufen, kommt es in wirklichkeit häufig vor, dass sich während der ausführung planungsfehler herausstellen oder während des gebrauchs ausführungsfehler und/oder planungsfehler. In diesen fällen sind rückschritte im entwicklungsprozess unvermeidlich.

Angemessenheitsproblem Ein allgemeines schema, das auf wirklich alle projekte passt (so wie das oben aufgeführte), hat fast keinen informationsgehalt mehr. Ein sehr detailliertes schema bietet viel information und hilfe

> bei der organisation, bringt aber die gefahr mit sich, dass es nur auf wenige projekte passt, vielleicht sogar nur ein einziges. In der tat hat jedes projekt in wirklichkeit seinen eigenen lebenslauf.

Geht man davon aus, dass vorgehensmodelle nur beschreiben, wie etwas im prinzip abläuft, dass man sie aber nicht allzu pedantisch ernst nehmen sollte, so sind sie ein ganz brauchbares mittel für die projektplanung und projektführung. Das erste in der SE-literatur dokumentierte vorgehensmodell stammt von Royce (1970); auf dieser basis wurde 1976 ein etwas detaillierteres schema von Barry W. Boehm vorgeschlagen (s. abb. 40); dieses bild ist auch als „waterfall chart“ bekannt.

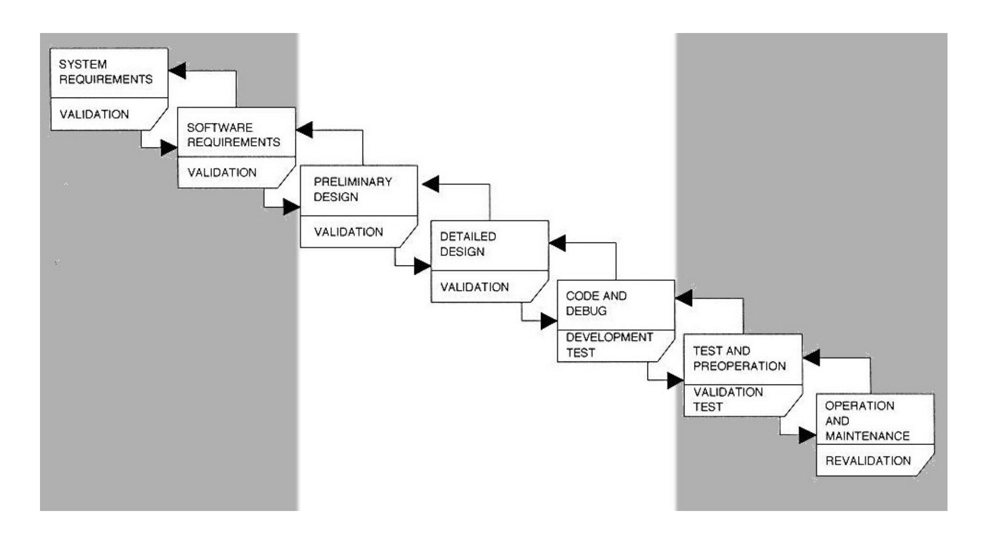


Abbildung 40: Wasserfallmodell nach Boehm

Hier wird der systementwicklungsprozess in sieben voneinander getrennte *phasen* zerlegt, die logisch aufeinander aufbauen, wobei in der tat entsprechend den zuvor gemachten bemerkungen unterschieden wird zwischen *system requirements* und *software requirements*. Als endergebnis jeder einzelnen phase können wir uns vorstellen, dass gewisse *dokumente* (spezifikationen, programme, testpläne etc.) abgeliefert werden. Das management spricht im zusammenhang mit den phasenübergängen sehr gerne von sog. *meilensteinen* (milestones). Diese meilensteine dienen zur strukturierung des gesamten entwicklungsprozesses, haben aber andererseits auch die aufgabe, die zuteilung der ressourcen festzulegen und den projektfortschritt zu ermitteln. Man nennt ein solches vorgehensmodell auch *dokumentengesteuert* oder dokumentenorientiert (document driven). In der fassung von Boehm wird zwischen den tätigkeiten und ihren ergebnissen nicht sauber unterschieden. Auch kommt der begriff des meilensteins hier nicht vor.

Ausserdem scheint das modell auch rückschritte explizit für jeweils eine phase und ihren vorgänger vorzusehen; liest man Boehm jedoch genauer, stellt man fest, dass er dies nur als einen *bezug* zur vorhergehenden phase für die zwecke der *validation* gedeutet haben möchte. Trotzdem wird Boehm von vielen leuten in der o.a. weise missverstanden. Bezeichnend ist in diesem zusammenhang, dass die meisten arbeiten, in denen Boehms waterfall chart zitiert wird, noch einen „grossen" rückwärtspfeil von *operation and maintenance* zurück zu *system requirements* ziehen. Solch ein pfeil würde sich natürlich via transitivität aus dem angegebenen falschen verständnis der „kleinen" rückwärtspfeile ergeben.

6.2 Kritik am wasserfallmodell

> *The management question, therefore, is not* whether *to build a pilot system and throw it away. You* will *do that. [. . .] Hence* plan to throw one away; you will, anyhow. *(Brooks 1975)*

> *. . . systems requirements cannot ever be stated fully in advance, not even in principle, because the user doesn't* know *them in advance — not even in principle. To assert otherwise is to ignore the fact that the development process itself changes the user's perceptions of what is possible, increases his or her insights into the applications environment, and indeed often changes that environment itself. We suggest an analogy with the Heisenberg Uncertainty Principle: any system development activity inevitably changes the environment out of which the need for the system arose. System development methodology* must *take into account that the user, and his or her needs and environment, change during the process. (McCracken und Jackson 1982)*

Seit beginn der 80er jahre wird ständig kritik am wasserfall-modell geäussert; die kritikpunkte sind im wesentlichen:

1. Dieses modell eignet sich nur für relativ einfache projekte, bei denen wir tatsächlich in einem wurf erfolg haben können.

2. Das frühe festschreiben der anforderungen ist sehr problematisch, da wir bereits gelernt haben, dass änderungen umso teurer werden, je weiter sie sich in die früheren phasen auswirken. Missverständnisse während der problemanalyse kommen viel zu spät ans tageslicht. Die behebung erfordert dann viel zu viel kosten; schlimmstenfalls wird das system gar nicht benutzt.

3. Die einführung des systems erfolgt erst sehr spät nach beginn des entwicklungszyklus. Dementsprechend beginnt der kapitalrückfluss (ROI = return on investment) erst sehr spät.

4. Die einführung des systems erfolgt auf einen schlag (big bang). Durch die vielzahl notwendiger änderungen im betriebsablauf kann eine organisation leicht überfordert werden. Es tauchen dann probleme mit der software auf, die im grunde genommen keine eigentlichen software-probleme, sondern vielmehr probleme der handhabung sind.

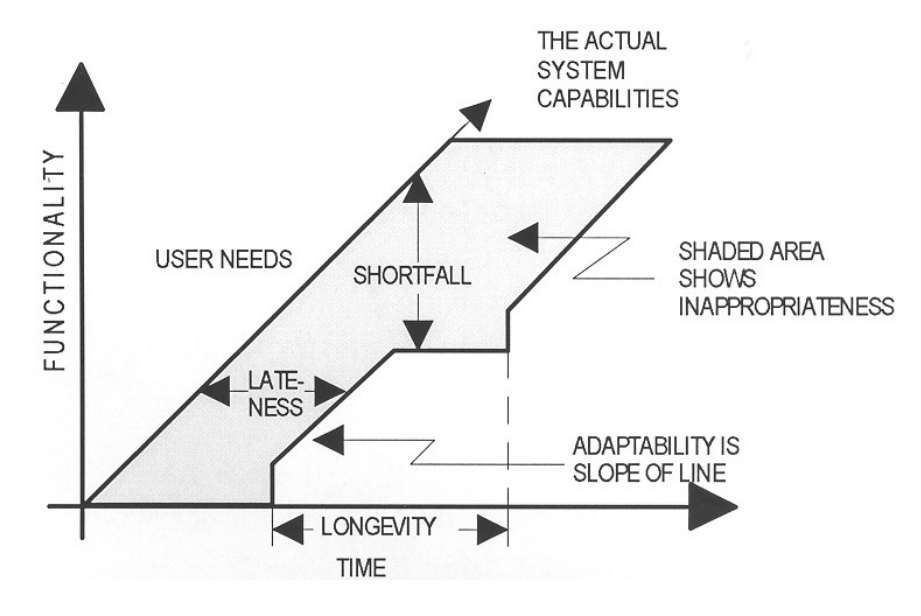


Abbildung 41: Produktivitätsmetriken für software-lebensläufe

Will man etwas eingehendere kritik an vorgehensmodellen wie dem wasserfallmodell üben, so sollte man die entwicklung der gewünschten und tatsächlich implementierten funktionalität eines systems in abhängigkeit von der zeit darstellen, wie dies in abbildung 41 geschehen ist. Diese darstellung stammt von Davis u. a. (1988). Auf der linken seite des schattierten bereichs finden wir die linie der benutzeranforderungen. Hier wurde angenommen, dass sich die anforderungen der benutzer an die funktionalität des systems über die zeit hinweg linear steigern. Wie weit diese annahme realistisch ist, mag dahingestellt sein. Man könnte ebenso gut mit einem exponentiellen wachstum der benutzeranforderungen rechnen, was die im folgenden darzustellende problematik noch verschärfen würde. Auf der rechten seite des schattierten bereichs finden wir die tatsächlichen fähigkeiten des systems zu einem gewissen zeitpunkt. Durch vergleich von gewünschter und vorhandener funktionalität in der horizontalen achse ergibt sich der verspätungsfaktor (lateness) des implementierten systems. Durch vergleich in der vertikalen achse ergibt sich das defizit (shortfall). Insgesamt zeigt der schattierte bereich das ausmass an, zu welchem das system unzufriedenstellend ist.

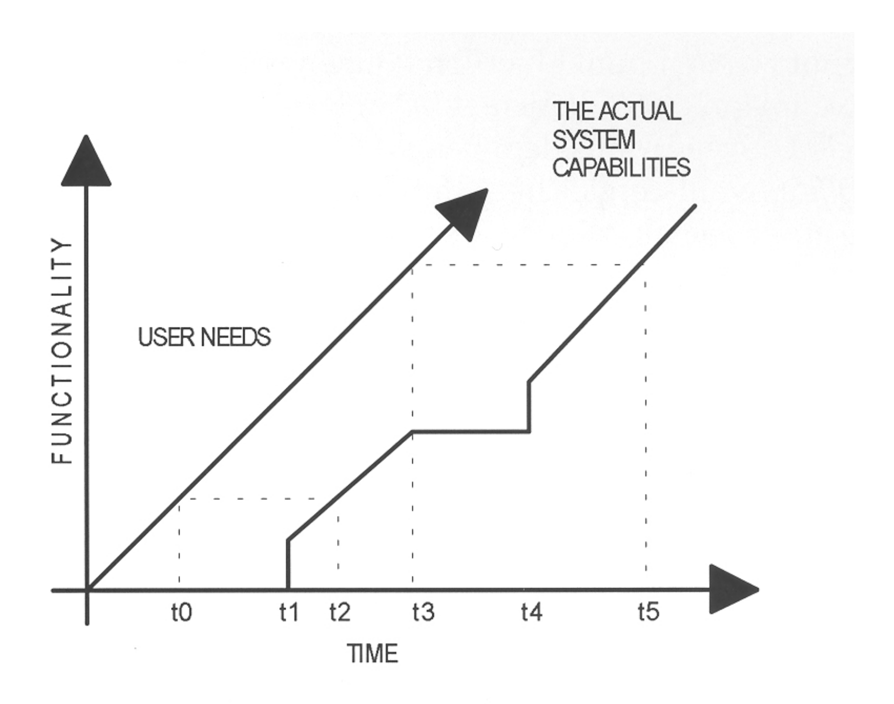


Abbildung 42: Produktivität des wasserfallmodells

Der übersichtlichkeit halber wollen wir die in dieser produktivitätsmetrik verwendeten begriffe nochmals zusammenstellen:

Defizit Ein defizit ist ein mass dafür, wie weit die funktionalität eines systems zu einem zeitpunkt t hinter den anforderungen zurückbleibt.

Verspätung Die verspätung ist ein mass für die zeit zwischen dem erkennen einer neuen anforderung und ihrer implementierung im system.

Änderbarkeit Änderbarkeit ist ein mass für die geschwindigkeit, mit der ein system an neue anforderungen angepasst werden kann, sie entspricht der steigung der funktionalitätslinie.

Langlebigkeit Die langlebigkeit gibt die zeit wieder, in der das system anpassungsfähig ist, das heisst also den zeitraum von der ersten entwicklung über die wartung bis zur ablösung durch ein neu entwickeltes system.

Unangemessenheit Die unangemessenheit wird durch die schattierte fläche zwischen den benutzeranforderungen und der systemfunktionalität dargestellt. Sie beschreibt somit den zeitlichen verlauf des defizits.

Ein wirklich ideales system hätte eine leere unangemessenheitsfläche, d.h. neue anforderungen wären unmittelbar erfüllbar.

swll.tex

Entsprechend dieser darstellung finden wir in abb. 42 die beurteilung der softwareentwicklung nach dem vorgehensmodell des wasserfalls. Zu einem zeitpunkt t0 sei hier die notwendigkeit einer softwarelösung erkannt worden und ein produktionsprozess nach dem wasserfallmodell begonnen. Zu einem zeitpunkt t1 wird das fertige system dann ausgeliefert; aufgrund der inhärenten missverständnisse bei der systemanalyse erfüllt es jedoch nicht einmal die ursprünglichen anforderungen zum zeitpunkt t0, geschweige denn die inzwischen gewachsenen anforderungen des benutzers. Vom zeitpunkt t1 bis t3 wird das produkt deshalb einer reihe von anpassungsarbeiten unterzogen, die man im landläufigen sprachgebrauch als wartung bezeichnet. Dabei wird zu einem zeitpunkt t2 tatsächlich die ursprünglich zum zeitpunkt t0 spezifizierte funktionalität erreicht. Zu einem zeitpunkt t3 stellt sich heraus, dass weitere wartungsarbeiten an diesem softwareprodukt nicht sinnvoll oder nicht möglich sind, es wird daher ein neues projekt begonnen. Wird dieses wiederum nach dem wasserfallmodell durchgeführt, so ergibt sich erneut eine verzögerung, bis zum zeitpunkt t4 wiederum ein fertiges system abgeliefert werden kann. Auch dieses erfüllt aber nicht die zum zeitpunkt t3 spezifizierten anforderungen, sodass sich die argumentation hier wiederholt.

Wir sehen, dass es praktisch niemals möglich ist, die anforderungen der benutzer auch nur annähernd zu erfüllen.

6.3 Das V-modell

Kaum besser als das wasserfallmodell ist das V-modell, das trotz der genannten schwächen immer noch das vorgeschriebene prozessmodell für alle IT-projekte der deutschen bundesbehörden inkl. der bundeswehr ist (Höhn und Höppner 2008). Im prinzip ist hier der wasserfall einfach in der mitte geknickt und teilweise nach oben umgelenkt, wodurch sich immerhin durch gedachte horizontale verbindungslinien ein bezug zwischen den phasen herstellen lässt. Ursprünglich als *V-Modell 92* publiziert und später aktualisiert zum *V-Modell 97*, grassiert es inzwischen unter der markenbezeichnung *V-Modell XT*, wobei „XT“ für „eXtreme Tailoring“, d. h. also eine extreme anpassbarkeit steht.

Abb. 43 zeigt das V-Modell XT mit den bundeswehr-spezifischen erweiterungen.

Zwei verfahrensweisen werden zur ablösung der software-entwicklung entsprechend einem software-lebenslauf vorgeschlagen; wir werden sie in den nächsten beiden abschnitten diskutieren: *frühe prototypen* und *inkrementeller bau*.

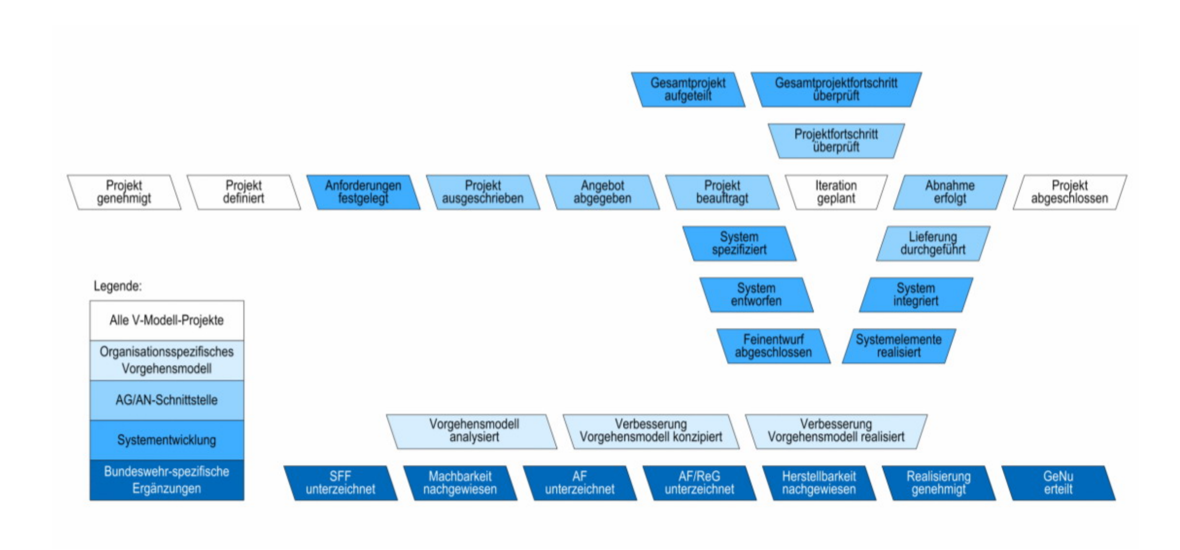


Abbildung 43: Das V-Modell XT

6.4 Frühe prototypen

Bei dieser strategie („rapid prototyping“) geht es darum, möglichst schnell einen oder mehrere prototypen eines systems zu gewinnen, um damit zu experimentieren und erfahrungen zu sammeln. Dabei benutzt man in möglichst grossem umfang bereits existierende software und attrappen; d.h. also module, die eine bestimmte funktion nur simulieren. (Analogie im automobilbau: windkanalmodell aus plastilin im massstab 1:1, designstudien, innenraumnachbauten, aufbau eines prototypen aus teilen anderer typen). Anhand der mit den prototypen gewonnenen erfahrungen wird anschliessend das system von grund auf neu entworfen, wobei dann ein modifizierter (d.h. verkürzter) lebenslauf befolgt wird.

Unterscheiden müssen wir bei prototyping zwischen sog. *explorativen prototypen*, die nur zur ermittlung von bestimmten vorstellungen der benutzer dienen bzw. die machbarkeit gewisser konzepte nachweisen sollen und die später keinen eingang in das fertige produkt finden („throwaway prototypes“) und sog. *evolutionären prototypen*, bei denen jeweils verbesserte versionen des systems hergestellt werden, die zunächst prototypisch einsetzbar sind, aber zu weiten teilen ihren eingang in das fertige produkt finden werden. Hier ist eine grenzlinie zum *inkrementellen bau* (s. nächster abschnitt) teilweise recht schwer zu ziehen.

Manche leute verwenden die scherzhafte bezeichnung *provotypen* für bestimmte explorative prototypen, die so konstruiert sind, dass sie reaktionen des benutzers provozieren.

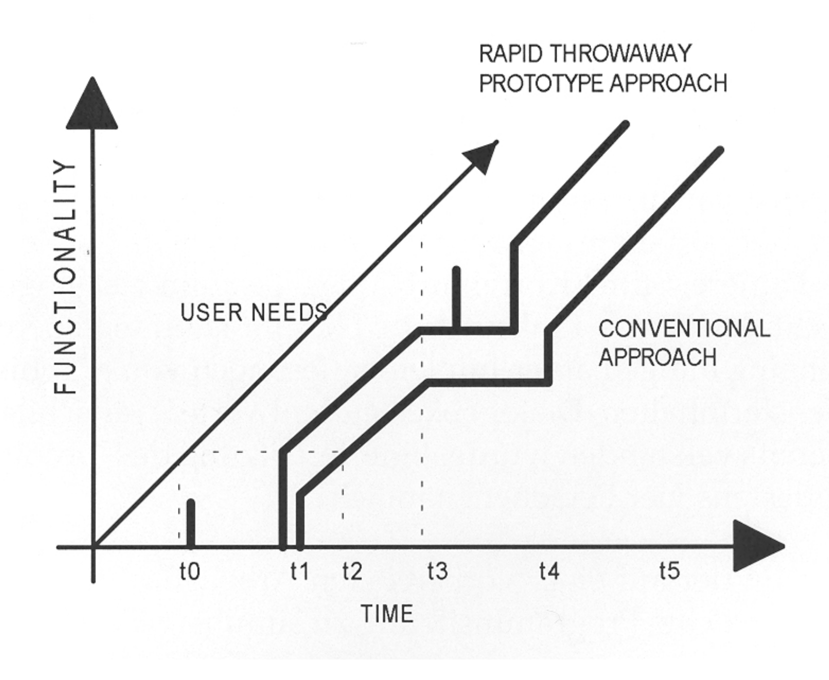


Abbildung 44: Wasserfallmodell mit explorativen prototypen

In abb. 44 ist die produktivität eines wasserfallmodells mit explorativen prototypen gegenübergestellt mit dem reinen wassermodell. Wir erkennen, dass bereits kurz nach dem beginn der systemtentwicklung zum zeitpunkt t0 ein prototyp hergestellt wurde, der nennenswerte teile der gewünschten funktionalität implementiert. Auf der basis dieses verbesserten verständnisses kann ein fertiges system bereits zu einem früheren zeitpunkt als t1 abgeliefert werden und erfüllt dann tatsächlich alle zum zeitpunkt t0 spezifizierten anforderungen. Jetzt besteht nur noch das problem, dass sich die anforderungen des benutzers in der zwischenzeit weiterentwickelt haben; über eine gewisse zeit hinweg kann man auch hier mit wartungsarbeiten den abstand zu den anforderungen gering halten. Irgendwann wird wiederum eine neuentwicklung notwendig, die dann wiederum einen explorativen prototypen verwendet. Abb. 44 ist insofern idealisiert, als nur jeweils ein prototyp erstellt wurde; in der praxis werden sicher häufig auch folgen von prototypen notwendig werden.

Bei der erstellung von prototypen müssen nicht unbedingt immer sehr teuere und aufwendige mittel verlangt werden. Eine interessante möglichkeit vor allem für die modellierung von benutzerschnittstellen, aber auch damit zusammenhängend für die erkundung der gewünschten systemfunktionalität stellen die von Rettig (1994) vorgeschlagenen „lo-fi prototypes“ dar. Die be-

zeichnung „lo-fi" ist hier allerdings ausgesprochen ungünstig gewählt. Gemeint ist sie als gegensatz zu Hi-Fi, welches in der unterhaltungsindustrie höchste wiedergabetreue bezeichnet und dementsprechend in der regel mit hohen preisen verbunden ist. Die lo-fi prototypes sind jedoch eher als „low-cost"-prototypen gemeint, weil ihre herstellung nicht übermässig grossen aufwand verlangt.

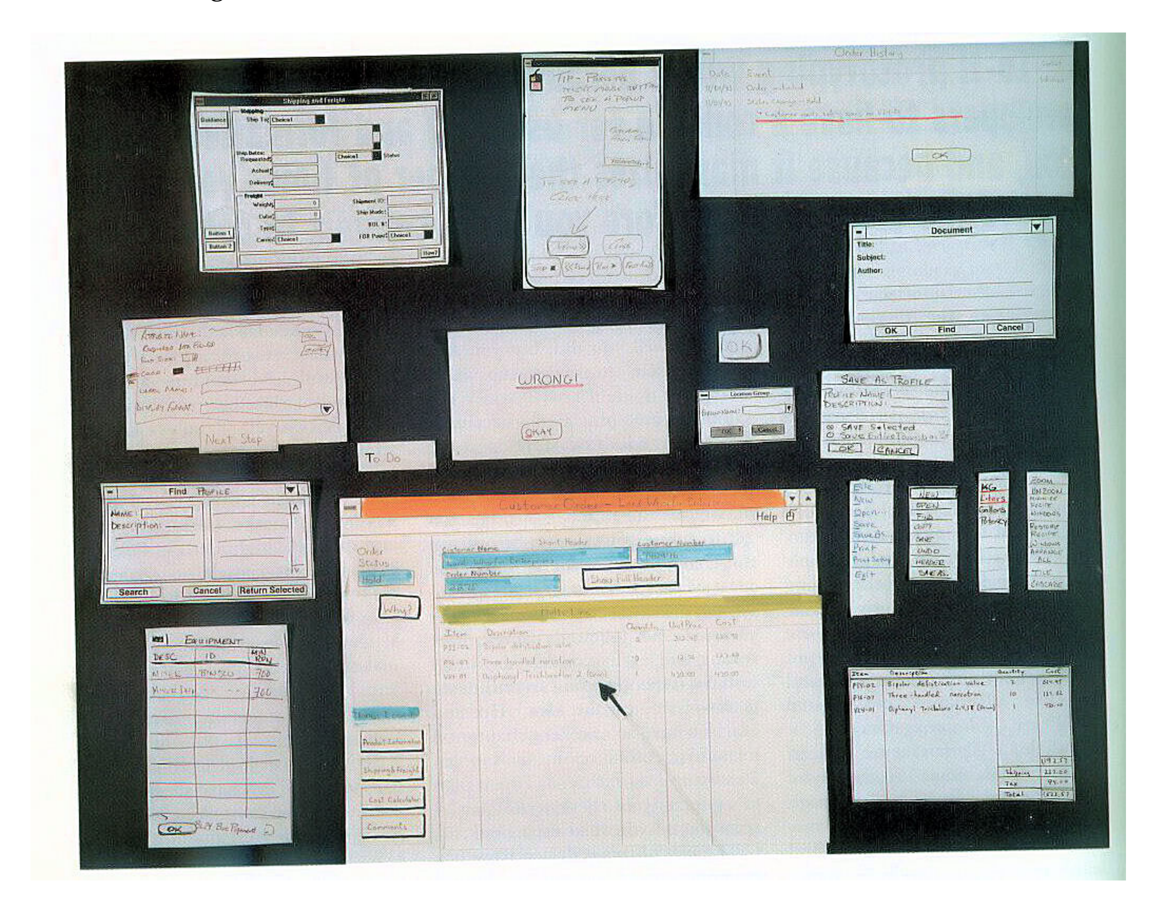


Abbildung 45: Material für die lo-fi prototypes

Als vorbereitung werden im fachgeschäft für zeichenbedarf allerlei farbige papiere, stifte, korrekturbänder, tesafilm etc. eingekauft (abb. 45); Rettig gibt eine recht detaillierte einkaufsliste. Hiermit kann man bereits vor der eigentlichen sitzung billige papier-implementierungen von graphischen widgets, menüs, fensterinhalten, dialogboxen etc. entwerfen, je nachdem, wie gut das problem bereits verstanden wurde. Eine testsitzung des prototypen involviert dann mindestens vier personen (abb. 46), nämlich

1. den benutzer, der auf einer vorbereiteten arbeitsfläche anhand der papierprototypen eine programmsitzung zu simulieren versucht,

2. einen eigenen betreuer (englisch: „facilitator"), der den benutzer durch

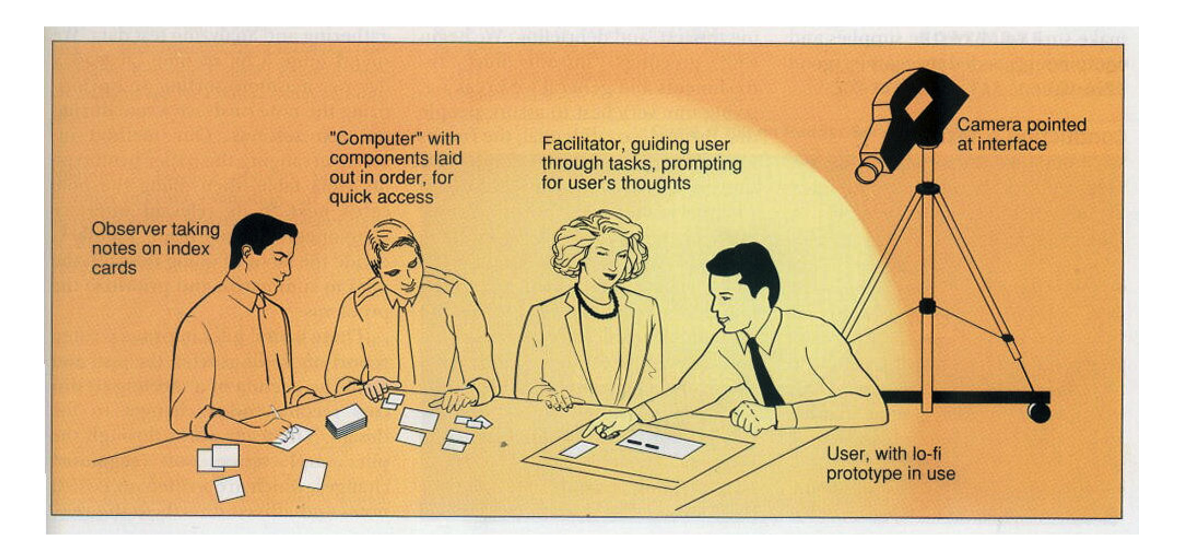


Abbildung 46: Lo-fi prototype setup

seine aufgabe führt, ihm fragen stellt und seine gedanken bei den arbeitsläufen zu erkunden sucht,

3. einen sog. „computer“, der in grosser geschwindigkeit mit den vorbereiteten kärtchen hantiert oder im bedarfsfall neue anfertigt und entsprechend den wünschen des benutzers die arbeitsfläche modifiziert und

4. einen beobachter, der sich über die ereignisse und beobachtungen notizen anfertigt.

Auch die notizen werden auf kärtchen notiert (jeweils eine beobachtung pro karte), sodass sie nach der sitzung problemlos nach sachgebieten geordnet werden können. Zur vermeidung irgendwelcher missverständnisse zeichnet eine videokamera alle vorgänge auf der arbeitsfläche und die dazu gesprochenen kommentare auf.

6.5 Inkrementeller bau

Beim inkrementellen bau versucht man zunächst, ein ganz schmales system mit ganz wenigen funktionen aufzubauen, damit sich der auftraggeber bzw. spätere benutzer an die handhabung des systems gewöhnen kann und rechtzeitig änderungswünsche anbringen kann. Nach und nach werden dann funktionen hinzugefügt.

Wichtig beim inkrementellen bau ist, dass von anfang an ein gewisses *minimalsystem* (basissystem) spezifiziert wird und dann eine folge sog. *inkremente*, die sich zum basissystem hinzufügen lassen. Als meilensteine wird man in

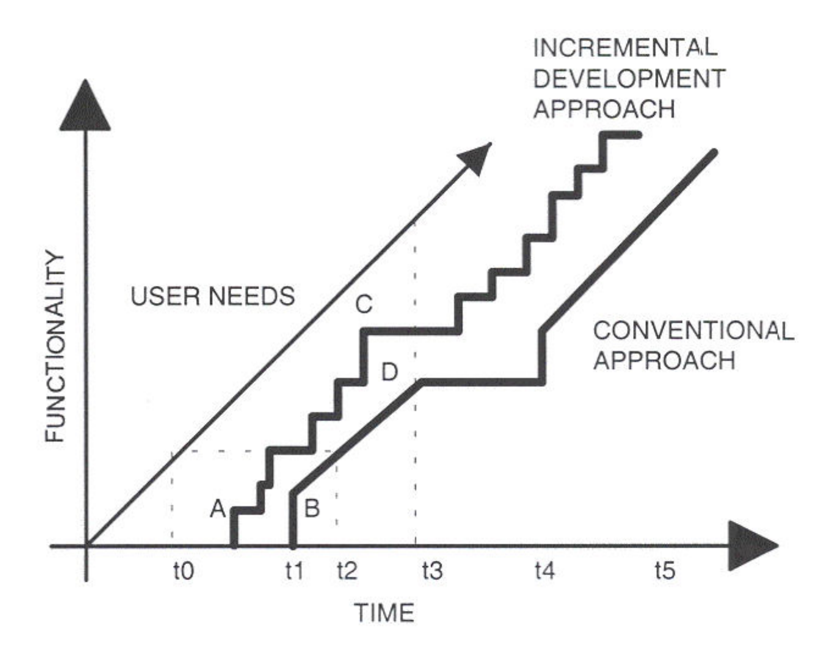


Abbildung 47: Produktivität des inkrementellen baus

einem solchen fall vermutlich die ablieferungszeitpunkte der inkremente definieren. Inkrementeller bau hat die folgenden vorteile:

- Die rückmeldung der benutzer während des betriebs der ersten systemstufen führt zu einem genaueren verständnis der anforderungen. Auf diese art und weise kann ein system erstellt werden, das den benutzeranforderungen besser gerecht wird (s. abb. 47: Der punkt C beim inkrementellen bau liegt in der funktionalitätsdimension höher als der punkt D des konventionellen ansatzes und ist überdies in der zeitachse früher angesiedelt.)

- Von der systemeinführung her ergibt sich der vorteil, dass die stufenweise einführung den big bang-effekt vermeidet.

- Der kapitalrückfluss beginnt früher, da erste systemstufen bereits nach kürzerer zeit verfügbar sind.

- Das gesamtsystem wird früher als gewöhnlich fertig, weil durch die rückkopplung mit dem benutzern vermeintlich wichtige teile als unwesentlich erkannt werden und dann nicht implementiert zu werden brauchen.

Dem stehen selbstverständlich auch gewisse nachteile gegenüber. Einige davon sind:

- Die reibungslose integration der inkremente hängt entscheidend von der gewählten architektur ab. Ändern sich die grundsätzlichen anforderungen im verlauf der entwicklung, so werden die nachfolgenden inkremente selbstverständlich entsprechend angepasst. Die gewählte systemarchitektur beruht aber auf den ursprünglich definierten anforderungen. Entweder hat man hier also ein inkongruentes system zu tolerieren oder es müssen bereits fertiggestellte inkremente neu entwickelt werden, was das modell wiederum eher in die nähe des klassischen wasserfallmodells bringt.
- Die mehrmalige einführung von verschiedenen inkrementen erhöht den organisatorischen aufwand und kann endbenutzer verwirren.

Frühe prototypen und inkrementeller bau sind schwer voneinander zu trennen: Normalerweise wird man auch beim inkrementellen bau das erste fertige system verwerfen müssen, um ein neues system mit denselben äusseren merkmalen von grund auf neu zu bauen. Inkrementeller bau ist auch nicht immer in beliebigem umfang möglich.

6.6 Agile methoden

Fussend auf der vielfach gemachten erfahrung, dass es praktisch unmöglich ist, ein grösseres softwareprojekt „in einem wurf" richtig durchzuführen, haben sich seit den späten 1990er jahren eine reihe von (objektorientierten) softwareprozessen durchgesetzt, die einerseits sehr stark auf interaktion mit den zukünftigen benutzern ausgerichtet sind und andererseits den stetigen wandel der eigentlichen projektziele in den vordergrund rücken. Zusammenfassend werden diese entwicklungsmethoden als *agile methoden* bezeichnet. Die zugrundeliegende erkenntnis ist nicht neu; in der tat gibt das zitat von McCracken und Jackson am anfang von abschnitt 6.2 diesen gedanken wieder. In einer wesentlich älteren fassung wurde er aber schon von dem architekten C. Alexander 1975 postuliert, der hier bereits bei der diskussion der objektorientierten muster angesprochen wurde und der geschrieben hat:

> *Es ist schlicht unmöglich, heute schon festzulegen, wie die umwelt in der zukunft aussehen soll, und dann den schrittweisen entwicklungsprozess so zu steuern, dass er dieses feste, imaginäre ziel erreicht.*

Ein früher vertreter dieser ansicht ist auch das von Christiane Floyd u. a. (1989) propagierte *STEPS (Softwaretechnik für evolutionäre, partizipative Softwareentwicklung)*, damals allerdings noch nicht als „agile methode" bezeichnet.

6.6.1 Extreme Programming

Eine sehr interessante entwicklung ist das von Kent Beck (1999) propagierte „Extreme Programming (XP)“. Kent Beck baut hier auf seiner langjährigen erfahrung im objektorientierten programmieren (in Smalltalk) auf. Auch er verzichtet darauf, einen grossen plan für das gesamtprojekt zu entwerfen, sondern es wird ganz stark inkrementell gearbeitet.

Beim XP-prozess arbeiten auftraggeber und programmierer die ganze zeit zusammen (wobei „auftraggeber“ in wirklichkeit ein oder mehrere *vertreter* des/der auftraggeber bedeutet). Der auftraggeber definiert funktionalitäten des systems in form von *user stories* (*szenarien*) und liefert gleichzeitig testfälle, anhand deren man die implementierung dieser szenarien überprüfen kann. Die programmierer liefern dann für jedes einzelne szenarium eine abschätzung, wie lange sie für dessen implementierung brauchen werden. Ergibt sich bei der abschätzung, dass die implementierung länger als drei wochen dauern wird, so muss der auftraggeber es in stücke aufteilen, die unterhalb dieser grenzen machbar sind. Es wird sodann ein plan für die nächsten eins bis drei wochen gemacht, denn in diesem takt werden systemversionen (*iterationen* genannt) hergestellt: Welche szenarien können für die nächste iteration des systems implementiert werden? In der regel wird es nicht möglich sein, alles zu implementieren; in diesem fall wählt der auftraggeber anhand der schätzungen der programmierer diejenigen szenarien aus, die für ihn eine so hohe priorität haben, dass er sie in der nächsten iteration verwirklicht sehen möchte. Programmierer dürfen nichts implementieren, das nicht ausdrücklich verlangt worden ist. Nach ablauf der maximal drei wochen wird gemeinsam evaluiert, ob das geplante erreicht worden ist und es wird die nächste iteration des systems geplant. Im takt von 1 bis 3 monaten werden releases an den kunden ausgeliefert; die programmierer selbst konsolidieren ihre änderungen an der codebasis jedoch mehrmals pro stunde (empfohlen wird ein viertelstundentakt).

Jeder im projekt hat das recht, jedes stück code zu verändern; dieses merkmal des prozesses zusammen mit dem häufigen „abhaken“ des bereits erreichten und dem weitgehenden verzicht auf spezifikationen und dokumentationen kommen ganz klar aus der *hackerkultur*. Dieser begriff wird heute hauptsächlich mit illegalen manipulationen an computern in verbindung gebracht; Cusumano und Selby (1995) weisen jedoch (in anderem zusammenhang, s. abschn. 7) darauf hin, dass der begriff *hacker* im ursprünglichen jargon der 60er jahre (siehe auch das *Hacker's Dictionary* von Raymond (1991)) ganz einfach personen bezeichnet, die sich mit irgendwelchen geräten, systemen oder computern bis in die letzten einzelheiten vertraut machen, ständig auf der suche nach verborgener funktionalität und möglichkeiten des kreativen einsatzes wenig genutzter vorrichtungen. Sie haben insofern grosse ähnlichkeit mit den

amateurfunkern, hobbyelektronikern, autobastlern etc.

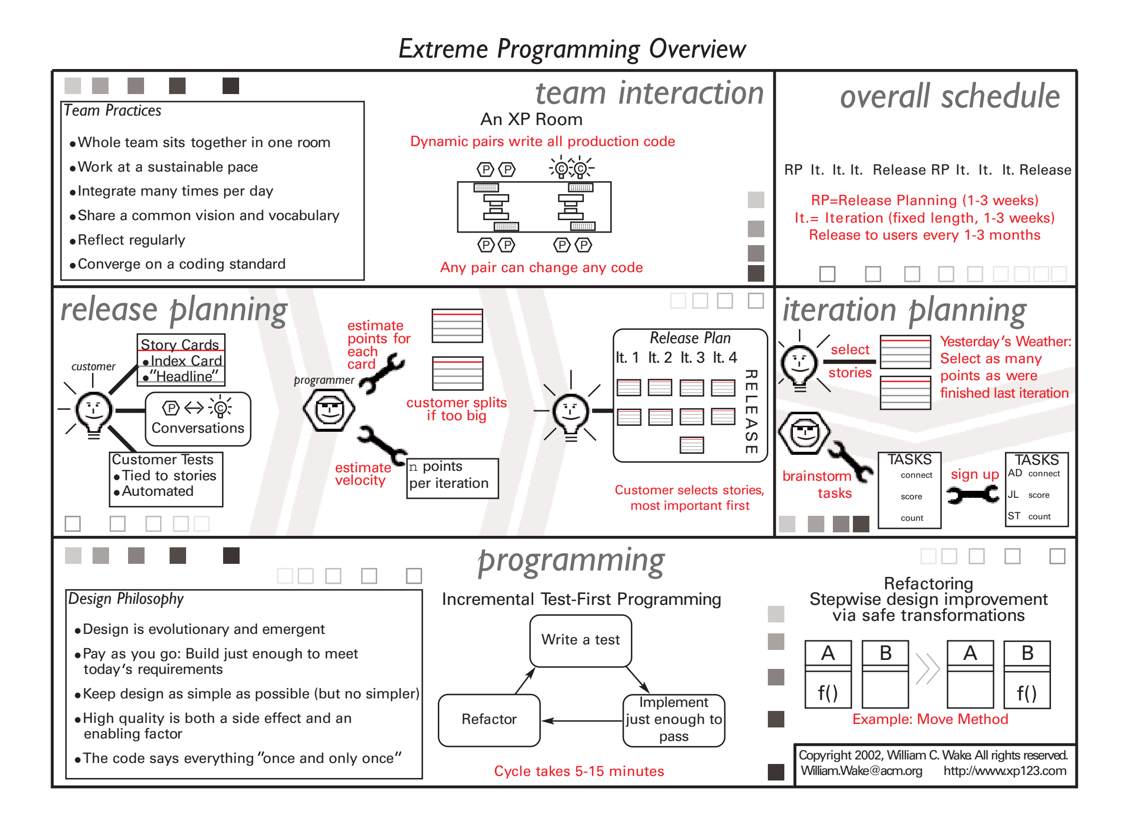


Abbildung 48: XP on one page

Eine weitere eigenart von XP ist das *pair programming*: Programmierer dürfen niemals alleine arbeiten, sondern sitzen immer zu zweit vor einem rechner. Auf diese art und weise soll jede einzelne codezeile bereits das resultat einer kritischen bewertung sein. Die grundregel bei XP heisst: *„Do the simplest thing that could possibly work.“* Es findet deshalb immer und immer wieder eine überarbeitung des schon geschriebenen codes statt, mit dem ziel, diesen zu vereinfachen. Im übrigen müssen die programmierer, *bevor* sie neuen code zum system hinzufügen, zuerst einen testfall schreiben (, der mit dem alten system natürlich fehlschlägt); der neue code muss diesen test bestehen.

Einen überblick über XP auf einer seite (von Bill Wake, `www.xp123.com`) bietet abb. 48.

In der ersten publizierten version von XP spricht Beck von 12 „Xtuden“[30],

[30]in anlehnung an die *etüden*, die er im gitarrenunterricht spielen musste.

die[31] wie folgt in vier kategorien eingeteilt werden; die einteilung findet sich im Wiki[32]. Der komplette englische text findet sich im anhang E:

- Feingranulare rückkopplung
  - *TDD – TestDrivenDevelopment* Testgetriebene entwicklung mit *unit tests* und *customer tests* (auch *akzeptanztests* genannt)
  - *PG – PlanningGame* Planungsspiel
  - *WT – WholeTeam* Ganzheitliche mannschaft
  - *PP – PairProgramming* Programmieren in paaren
- Stetiger prozess statt schubweise bearbeitung
  - *CI – ContinousIntegration* Stetige integration
  - *DI – DesignImprovement* Entwurfsverbesserung
  - *SR – SmallReleases* Kleine releases
- Gemeinsames verständnis
  - *SD – SimpleDesign* Einfacher entwurf *(DoSimpleThings, YouArentGonnaNeedIt, OnceAndOnlyOnce, SimplifyVigorously)*
  - *SM – SystemMetaphor* System-metapher
  - *CCO – CollectiveCodeOwnership* Gemeinsames eigentum am code
  - *CS – CodingStandard* Programmiernorm oder programmierkonventionen
- Wohlergehen der programmierer
  - *SP – SustainablePace* Nachhaltige geschwindigkeit

Von der *tests first*-devise sprachen wir schon. Dabei sind die *unit tests* diejenigen testfälle, welche von den programmierern geschrieben werden, bevor sie code implementieren (und es sollte an dieser stelle bemerkt werden, dass alle diese testfälle aufgehoben werden und als *regressionstests* spätestens vor jedem release der software erneut durchgeführt werden). Die *akzeptanztests* sind die vom auftraggeber formulierten testfälle.

Der begriff *planungsspiel* soll einerseits den emotionalen stress im planungsprozess reduzieren und andererseits klarmachen, dass es hier genaue regeln gibt, an die sich jeder „spieler“ halten muss. Genaueres findet sich im Wiki an der genannten stelle.

[31]auf der Seite `http://c2.com/cgi/wiki?ExtremeProgrammingCorePractices`

[32]`http://c2.com/cgi/wiki`: The ideas of “Wiki” may seem strange at first, but dive in and explore its links. Wiki is a composition system, it’s a discussion medium, it’s a repository, it’s a mail system, it’s a tool for collaboration. Really, we don’t know quite what it is, but it’s a fun way of communicating asynchronously across the network.

„Ganzheitliches team“ bezieht sich auf die tatsache, dass immer ein vertreter des auftraggebers anwesend ist; über das programmieren in paaren sprachen wir schon.

Besonderes augenmerk verdient der punkt „entwurfsverbesserung“, der auch unter der überschrift „Gnadenlos refaktorisieren“ bekannt ist. Es heisst hier konkret, wenn man zwei methoden findet, die gleich aussehen, dann sind diese auszufaktorisieren („auszuklammern“) und durch eine einzige methode zu ersetzen. Gleiches gilt für zwei objekte mit gleicher funktionalität.

Interessant an dem ansatz ist, dass das wohlergehen der programmierer hier ausdrücklich betont wird. Der begriff der *nachhaltigkeit* (engl.: *sustainability*, was auch als „erträglichkeit“ übersetzt werden kann) geht in diesem zusammenhang auf die oft gemachte erfahrung zurück, dass in späten nacht- oder frühen morgenstunden oder überhaupt in mehr oder weniger unwillig geleisteten überstunden hergestellte softwareprodukte in aller regel nicht das gezahlte geld wert sind.

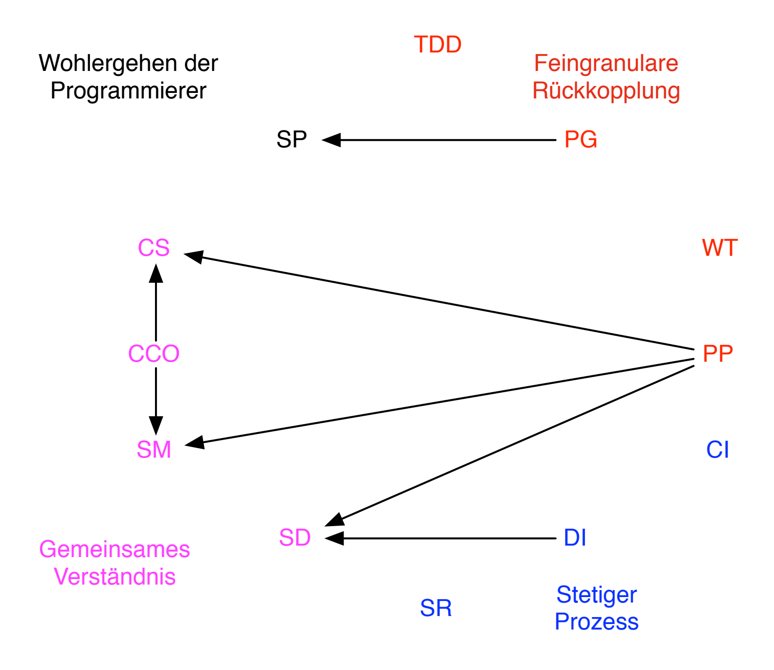


Abbildung 49: Einflüsse der Xtuden

Manche der Xtuden beeinflussen andere positiv; diese wirkungen sind in abb. 49 dargestellt.

Die rechte des softwareentwicklers im XP-prozess sind wie folgt beschrieben:

Rechte des Softwareentwicklers Du hast das recht

- zu wissen, was benötigt wird, mit klaren festlegungen von prioritäten in form detaillierter anforderungen und spezifikationen. Beim Extreme Programming kann der kunde die anforderungen und spezifikationen ändern, indem er *stories* hinzufügt oder entfernt, oder er kann die priorität der *stories ändern*. Der programmierer bekommt die neuen *stories* zu sehen, schätzt sie ab und informiert den kunden über die auswirkungen auf den arbeitsplan. Wenn der plan nicht mehr auf das gewünschte lieferdatum des kunden passt, informiert der entwickler den kunden und der kunde kann entweder den neuen plan akzeptieren oder genügend viele *stories* niedrigerer priorität entfernen, um das datum einzuhalten. Es ist keine katastrophe, dass der kunde änderungen vornimmt, sondern es ist unvermeidlich.
- klare und fortgesetzte kommunikation mit dem kunden zu haben, und zwar sowohl dem endbenutzer als auch der verantwortlichen autorität.
- jederzeit qualitätsarbeit zu leisten und unterstützung dafür zu finden, auch wenn das etwas länger dauert und den einkauf von werkzeugen erfordert.
- hilfe von kollegen, vorgesetzten und auftraggebern zu erbitten und zu bekommen, und dass gelegenheiten für die kommunikation mit anderen mitarbeitern des projekts im zeitplan vorgesehen sind.
- deine eigenen abschätzungen zu machen und anzupassen; das schliesst die beschaffung von daten für deine langfristigen pläne und ziele ein.
- die verantwortlichkeit für deine tagespläne und -ziele selbst zu haben.
- die unterstützung deiner vorgesetzten und kunden zu haben für eine fortlaufende fortbildung, einschliesslich — aber nicht beschränkt auf — bücher, abonnements, zeit und geld zum ausprobieren neuer programmierwerkzeuge, besprechungen, ausbildung etc.
- auf ein nachhaltiges arbeitstempo (eine vierzig-stunden-woche)
- deine eigenen entwicklungswerkzeuge, wo immer passend, einzusetzen, so lange das arbeitsergebnis verträglich ist mit den erwartungen deines kunden.

(`http://c2.com/cgi/wiki?DeveloperBillOfRights`, Oktober 2008)

Dem gegenüber stehen die

Pflichten des softwareentwicklers Du hast die pflicht

- zu verstehen, was benötigt wird und warum es die priorität deines auftraggebers ist.

- Qualitätsarbeit abzuliefern, auch wenn man dich nötigt, das gegenteil zu tun.

- andere zu betreuen und alle fertigkeiten und kenntnis, die du vielleicht hast, zu teilen.
- deine abschätzungen so genau wie möglich zu machen; zu wissen, wenn du hinter dem plan zurückbleibst, woran das liegt, und wie du den plan möglichst schnell so ändern kannst, dass er der realität entspricht.
- die konsequenzen deiner aktionen zu tragen.

(`http://c2.com/cgi/wiki?DeveloperBillOfResponsibilities`, Aug. 2006)

Auch der auftraggeber hat rechte:

Rechte des auftraggebers Du hast das recht

- die priorität einer jeden *user story* für dein unternehmen festzulegen.
- einen überblicksplan zu bekommen, zu wissen, was erreicht werden kann, wann, und zu welchen kosten.
- aus jeder programmierwoche den maximalen gegenwert zu erhalten.
- fortschritt in einem laufenden system zu sehen, das nachweislich funktioniert, indem es wiederholbare tests besteht, die du spezifizierst.
- entwickler über änderungen in den anforderungen zu beraten
- über zeitplanänderungen so früh informiert zu werden, dass du entscheiden kannst, wie die funktionalität reduziert werden kann, um das ursprüngliche datum wieder zu erreichen.
- jederzeit das projekt zu beenden und ein nützliches system zu behalten, das deinen investitionen bis zu diesem zeitpunkt entspricht.

(`http://c2.com/cgi/wiki?CustomerBillOfRights`, März 2010)

6.6.2 Refactoring

Wegen der wichtigkeit des begriffs „refactoring“ soll diesem ein eigener abschnitt gewidmet werden. Dabei beziehen wir uns auf Fowler u. a. (1999). Diese definieren *refactoring* als

> Eine änderung der internen struktur von software mit dem ziel, sie leichter verständlich oder billiger modifizierbar zu machen, ohne dabei ihr beobachtbares verhalten zu ändern.

Damit ist klar, dass *refactoring* niemals damit zu tun haben kann, neue funktionalität einzuführen; auch sind nicht irgendwelche beliebigen „aufräumarbeiten“ am code gemeint, sondern ganz gezielt solche massnahmen, die den

code einfacher und verständlicher machen. Die stellen, an denen angegriffen wird, sind von Kent Beck als „bad code smells“ (schwäbisch würde man sagen: „Das programm hat ein g'schmäckle.“) bezeichnet worden. Beispiele für solche *bad smells* sind:

Duplicated code Hier geht es nicht nur um code, der 1:1 dupliziert wurde, sondern auch um code, der an mehreren stellen in ganz ähnlicher form auftritt. Hier drängt sich die vermutung auf, dass eine modifikation dieses code an einer stelle auch eine modifikation an den jeweils anderen stellen zur folge haben müsste. Dann ist die gefahr jedoch gross, dass diese nachgeordneten modifikationen vergessen werden. Deshalb soll so ein redunanter code herausfaktorisiert werden, wobei die u. u. vorhandenen unterschiede durch parameter erfasst werden.

Long method Ist eine methode viel länger als die meisten anderen methoden, so erschwert dies das verständnis. Deshalb ist es sinnvoll, nach wegen zu suchen, diese methode in mehrere kleine methoden aufzuspalten.

Large class Eine ähnliche überlegung in bezug auf klassen.

Shotgun surgery „Schrotflinten-operation“: Hier ist ein konzept derart auf viele klassen verteilt, dass eine änderung des zugrundeliegenden konzepts änderungen in allen beteiligten klassen verlangen würde. Ziel ist es deshalb, diese klassen so neu zu bündeln, dass die zu dem konzept gehörigen codestücke möglichst innerhalb einer einzelnen klasse liegen.

Feature envy „Merkmalsneid“ liegt vor, wenn eine methode mehr auf die daten anderer klassen als auf die der eigenen zugreift. Das kann dadurch behoben werden, dass die methode in die andere klasse verlagert wird.

Wichtig beim *refactoring* ist, dass jeweils nur eine einzige massnahme in einem schritt vorgenommen wird und der code jeweils vor und nach dem *refactoring* die gleichen tests durchläuft. Nur so kann gewährleistet werden, dass das *refactoring* die bedeutung des programms erhält.

Beispiele für *refactoring*-operationen wären etwa das umbenennen von methoden, attributen, klassen etc. (in unter/oberklassen oder in völlig andere klassen), das verschieben von entitäten in andere klassen, das ändern einer methoden-signatur, das falten von konstanten etc. Viele werkzeuge unterstützen *refactoring*-prozesse.

6.6.3 Scrum

Eine weit verbreitete alternative zum XP-prozess ist der von Jeff Sutherland (`scrum.jeffsutherland.com`) propagierte *Scrum*-prozess (engl. scrum = „Ge-

Scrum Cheat Sheet



Product Owner
Owns the Product Backlog

The Product Owner represents the interests of everyone with a stake in the project (Stakeholder) and he is responsible for the final product.

- elicit product requirements
- manage the Product Backlog
- manage the release plan
- manage the Return on Investment

Scrum Master
Owns the Scrum process

The Scrum Master is responsible for the Scrum process. He ensures everybody plays by the rules. He also removes impediments for the Team. The Scrum Master is not part of the Team.

- manage the Scrum process
- remove impediments
- facilitate communication

Team Member
Owns the software

The team figures out how to turn the Product Backlog into an increment of functionality within a Sprint. Each team member is jointly responsible for the success of each iteration and of the project as a whole.

- software quality
- technical implementation of User Stories
- delivery of functional software increment
- to organize themselves

Requirements
Make **SMART** Requirements: Simple, Measurable, Achievable, Realistic, Traceable.

User Stories
INVEST in User Stores: Independant, Negotiable, Valuable, Estimatable, Small, Traceable.

Tasks
Make sure a Task is **TECH**. Time boxed, Everybody (can pick it up), Complete and Human-readable.

Sprint Planning
Commit the deliverable(s) to the PO

Two part meeting. First, the PO presents the User Stories. Second, when the Team thinks they have enough Stories to start the Sprint, they begin breaking it down in Tasks to fill the Sprint Backlog.

Timebox: 4 hours
Owner: Product Owner
Participants: Team, Scrum Master

Daily Scrum
Inspect and Adapt the progress

In this standup meeting the Team daily inspects their progress in relation to the Planning by using the Burndown Chart, and makes adjustments as necessary.
Timebox: 15 minutes

Owner: Scrum Master
Participants: Team, all interested parties may silently attend.

Sprint Review
Demonstrate the achievements

The team shows the PO the result -the potential shippable product- of the Sprint. The PO can accepts or rejects features depending on the agreed acceptance criteria.

Timebox: 2 hours
Owner: Team
Participants: Scrum Master, Product Owner

Retrospective
Maintain the good, get rid of the bad

At the end of a Sprint, the Team evaluates the finished Sprint. They capture positive ways as a best practice, identify challenges and develop strategies for improvements.

Timebox: 2 hours
Owner: Scrum Master
Participants: Team, Product Owner, optionally the PO can invite Stakeholders

Product Backlog
Dynamic prioritized list of requirements

The requirements for the product are listed in the Product Backlog. It is an always changing, dynamically prioritized list of requirements ordered by Business Value. Requirements are broken down into User Stories by the PO.

Prioritize the requirements by playing the Business Value game.

Buy these at www.agile42.com

Burndown Chart
Estimated remaining time of the Sprint

The Burndown chart shows the amount of work remaining per Sprint. It is a very useful way of visualizing the correlation between work remaining at any point in time and the progress of the Team(s).

Use a tool such as Agilo to automatically create the Burndown Chart.

Learn more at www.agile42.com

Sprint Backlog
List of committed User Stories

The Sprint Backlog contains all the committed User Stories for the current Sprint broken down into Tasks by the Team. All items on the Sprint Backlog should be developed, tested, documented and integrated to fullfil the commitment.

Estimate Story complexity by playing Planning Poker.

Buy these at www.agile42.com



Potential Shippable Product

Scrum requires at the end of each Sprint that the product is potential shippable to the customer. That means the increment is:

- thoroughly tested
- well-structured
- well-written code
- user operation of the functionality is documented

Abbildung 50: Scrum cheat sheet

dränge“), der in abb. 50 zusammengefasst ist. Mit dem XP-prozess hat er die kleinteilige planung und die häufigen releases gemeinsam; was in XP eine iteration ist, heisst hier *sprint*. Davon abgesehen, gibt es grosse unterschiede zum XP-prozess, insbesondere sind hier zwei rollen hervorgehoben:

Product owner Dieser definiert die übergreifenden ziele des projekts und kümmert sich um den *return on investment*.

Scrum master Dieser gehört nicht zum entwicklerteam, leitet aber den gesamten prozess und sorgt für die einhaltung der spielregeln.

Lesens- und beachtenswert ist, was hier zu den anforderungen, user stories und aufgaben gesagt wird:

- Make **SMART** requirements: Simple, Measurable, Achievable, Realistic, Traceable
- **INVEST** in user stories: Independent, Negotiable, Valuable, Estimatable, Small, Traceable
- Make sure a task is **TECH**: Time boxed, Everybody can pick it up, Complete, Human-readable.

6.6.4 Andere agile methoden

Von Alistair Cockburn (`alistair.cockburn.us`) stammt die *Crystal family* von methoden (Crystal Clear, Crystal Yellow, Crystal Orange, Crystal Red, Crystal Magenta, Crystal Blue) mit zunehmendem komplexitätsgrad. Jeff de Luca (`www.nebulon.com/articles/fdd/latestfdd.html`) schlug das *Feature Driven Development* vor, das eine gewisse ähnlichkeit zu der im nächsten kapitel zu schildernden Microsoft-methode hat.

Kontrollfragen

1. Erklären sie die drei grundlegenden, mit jedem vorgehensmodell verbundenen probleme!
2. Worin liegt die spezifische kritik am wasserfallmodell?
3. Wie definiert man produktivitätsmaße für vorgehensmodelle und was sind die relevanten größen?
4. Vergleichen sie die produktivitätsmetriken des wasserfallmodells mit denen der frühen prototypen und des inkrementellen baus!

`swll.tex`

5. Beschreiben sie die 12 kernpraktiken des *Extreme Programming*!
6. Was sind *bad code smells*? Was versteht man unter *refactoring*?

7 Die Microsoft-methode

Als kontrastprogramm zu den lebenslaufmodellen, die aus der klassischen softwaretechnik stammen, mag es interessant sein, die methoden des marktführers von PC-software zu studieren. Hier muss man allerdings zunächst grundlegend bemerken, dass diese software eine ganz andere natur hat als das, was wir im vorigen kapitel diskutiert haben: Einen auftraggeber, mit dem der inhalt des projekts zunächst erarbeitet und spezifiziert werden muss, gibt es hier nicht: wir haben es mit konfektionsware zu tun („*shrink-wrap software*"), die aufgrund eines antizipierten interesses grosser kundenkreise in standardisierter form vorfabriziert wird. Einige zitate von einer software-engineering-tagung, die ich aufgeschnappt habe, mögen die denkweise dieses industriezweigs beleuchten[33]:

1. *Shrink wrap software has its own laws.* All die fertig konfektionierte Software, die in Großmärkten im Regal steht — so genannt wegen ihrer Verpackung im Schrumpfschlauch, durch dessen Aufschneiden man automatisch die Lizenzbedingungen akzeptiert —, hat ihre eigenen Gesetze. Mag der Kunde bei individuell für ihn angefertigter Software, ähnlich wie bei jedem Werkvertrag, noch ein Recht darauf haben, daß das Produkt genau das von ihm beschriebene Problem löst, oder aber nachgebessert, zurückgenommen oder doch mit einem deutlichen Preisabschlag versehen werden muß, so existiert bei *shrink wrap software* (etwas professioneller als „COTS Software": *Commercial off the shelf* bezeichnet) kein wirklicher Vertrag zwischen Programmierer und Käufer. Die Leistungen des in bunter, ansprechender Verpackung angebotenen Programms werden eher in großen Zügen und jedenfalls mit deutlichen Zügen von (in der Werbung allgemein üblicher) Übertreibung beschrieben. Dementsprechend bewerten die Kunden dann auch das Resultat ihres Kaufs. Außerdem haben sie ja bereits durch das Öffnen der Verpackung die „Garantie"-Bedingungen akzeptiert, die in aller Regel nur versprechen, daß die CDs rund, silberfarbig und mit maschinenlesbaren Aufzeichnungen versehen sind. Darüber hinaus wird jede Verantwortung abgelehnt.
2. *PC software is a way of life.* Für die PC-Kunden ist Software nicht einfach Software, sondern Bestandteil ihres Lebensstils, ihrer Weltanschauung. So wie es in den fünfziger Jahren wichtig war, immer das neueste Auto vor dem Haus stehen zu haben, samstags stolz zu putzen und sonntags auszufahren — alles nur, um

[33] Die folgenden abschnitte sind ein ausschnitt aus Klaeren (2006)

die Nachbarn zu beeindrucken — so ist es bei PC-Software einfach schick, immer als erster die allerneueste Version aller möglichen Programme zu besitzen und abends in der Kneipe all die armen Leute zu bedauern, welche diese entweder noch nicht haben oder noch nicht beherrschen oder — man stelle sich das vor! — noch gar nicht gewußt haben, daß es schon wieder eine neue Version gibt. Das paßt natürlich vorzüglich zu dem umsatzfördernden Prinzip, dauernd neue *releases* zu Softwareprodukten herauszugeben. Übrigens ist die Industrie seit der praktisch ubiquitären Verbreitung von Internet-Anschlüssen den armen Ahnungslosen bereits weitgehend entgegengekommen, sie brauchen sich jetzt nicht mehr bei ihren Freunden zu blamieren: Praktisch jedes Programm macht bei seinem Start eine Verbindung zu seinem Produzenten auf und fragt dort nach neuen Versionen. Der Benutzer wird sogleich informiert und kann in aller Regel durch einen einfachen Mausklick die neue Version kaufen — vorausgesetzt, er hat seine Kreditkartendaten bzw. seine Bankverbindung damals bei der Registrierung der Software gleich hinterlegt.

3. *Get your team into ship mode.* Etwas frei übersetzt: „Mach deinen Leuten klar, daß die nächste Version jetzt verkauft werden muß.“ Das heißt also, nicht lange herumbasteln, verbessern und überlegen, ohne daß auch wieder Einkünfte generiert werden: Wir programmieren, um zu verkaufen, und für keinen anderen Zweck!
4. *Time to market is more important than product quality.* Wiederum frei übersetzt: Eine frühe Markteinführung ist wichtiger als ein perfektes Produkt. Das bessere Produkt kann nachgeschoben werden, Hauptsache, der Markt ist erst einmal besetzt. Hier stellt es sich als vorteilhaft heraus, daß Programme im Gegensatz etwa zu Autos, Fahrrädern, Waschmaschinen etc. einen signifikanten Lernprozeß bei ihrem Verwender verlangen. Es ist nicht ganz einfach, von einem System auf ein anderes umzusteigen: Einerseits riskiert man, alle seine vorherigen Arbeiten (Texte, Bilder, Filme, ...) zu verlieren, aber vor allem muß man sich selbst wieder auf ein völlig neues System „umschulen“, was man wirklich nur dann tut, wenn es ganz wesentliche Vorteile bringt.

Die methoden des marktführers Microsoft wurden von Cusumano und Selby (1995, 1996) eingehend studiert und publiziert, aber auch in einem zeitschriftenartikel (1997) zusammengefasst.

Allgemein wird festgehalten, dass die firma Microsoft hochgradig flexibel und auf unternehmerisches risiko ausgerichtet ist. Die Halbwertszeit von pro-

grammcode im haus Microsoft beträgt nur 18 monate (Cusumano und Selby 1995, p. 215). Microsoft unterscheidet sich damit von eher konservativen firmen, die längere produktzyklen und kompliziertere verwaltungsstrukturen besitzen. Microsoft hat die weisheiten der softwaretechnik nur wenig zur kenntnis genommen und statt dessen viele elemente der hackerkultur bewahrt. Das „glasnost-prinzip“ („jeder darf jedes stück code einsehen“) erwähnte ich bereits; so extrem wie beim XP („jedem gehört jedes stück code“) ist die hackerkultur dann doch wiederum bei Microsoft nicht etabliert.

Die softwareherstellung bei Microsoft ist in ungewöhnlich flacher hierarchie angeordnet; wir haben es im wesentlichen mit lose strukturierten kleinen teams von je 3–8 entwicklern zu tun, die parallel arbeiten, von einer gleich grossen gruppe von softwaretestern begleitet werden und einen chef haben. Ein solches team ist jeweils für eine bestimmte *vorrichtung* (*feature*) zuständig und hat die grösstmögliche autonomie für diese vorrichtung sowie die gesamtverantwortung vom entwurf über die programmierung bis zur wartung für diese vorrichtung. Cusumano und Selby bemerken, dass diese konstruktion der *feature groups* es im prinzip erlaubt, grosse teams von programmierern mit den gleichen methoden zu managen wie kleine teams. Das kann man durchaus auch anthropologisch deuten, denn entwicklungsgeschichtlich ist der mensch ein kleingruppenjäger und fühlt sich bis zum heutigen tag in kleineren gruppen wohler als in grossen, anonymen organisationen. Die heute vielfach beklagte „staatsverdrossenheit“ hat sicherlich ihre ursachen auch darin, dass „der einzelne“ sich gegenüber „dem system“ zunehmend als hilflos empfindet, und grosse firmen mit einer starken organisations-hierarchie verzeichnen häufig ähnliche symptome („innere kündigung“) bei ihren mitarbeitern. Dem wirkt die Microsoft-organisation eher entgegen; in einer *feature group* kann sich der einzelne wesentlich besser artikulieren als in einer komplexen organisation.

Grosses gewicht legt Microsoft nach Cusumano und Selby darauf, nur die besten leute einzustellen, wobei sie (mit einem schmunzeln zwischen den zeilen) bemerken, dass es unter umständen schon genügt, wenn alle *glauben*, sie seien eingestellt worden, weil sie die besten seien und dass jedenfalls alle anderen im team die besten sind.

Als vorsichtige wissenschaftler betonen Cusumano und Selby, dass es nicht behauptet wird, dass die im weiteren geschilderten methoden von Microsoft erfunden wurden, noch dass es diese methoden sind, die man als ursache für den erfolg von Microsoft bezeichnen würde.

Als kerngedanken stellen Cusumano und Selby eine methode heraus, die sie selbst die *sync and stabilise*-methode nennen und die sich in noch extremerer form beim XP-prozess findet. Täglich wird das gesamte softwareprodukt aus den möglicherweise modifizierten quelltexten erneut hergestellt und ge-

gen eine regressionsdatenbank von bekannten fehlern automatisch getestet. Dieser prozess heisst täglicher bau (*daily build*) bzw. nächtlicher bau (*nightly build*), da dieser vorgang in der regel über nacht stattfindet. Für diesen prozess ist in jedem projekt ein eigener mitarbeiter verantwortlich, der sogenannte *build master*. Die entwickler holen sich aus einer zentralisierten datenbank die programmquelltexte ab in einen privaten dateibereich, editieren diese, testen sie selbst und stellen sie in die zentrale datenbank zurück. Dieses zurückstellen geschieht nicht unbedingt täglich, aber in der regel wenigstens 2 mal pro woche. Das ergebnis nennt man eine *zwischenversion*. Der bis zu einem bestimmten zeitpunkt jedes tages in die datenbank zurückgestellte code muss compilierbar und bindbar sein und darf im fertigen produkt keine fehler hervorrufen. Normalerweise hat Microsoft also auch bei einer völlig im fluss befindlichen produktentwicklung jeden morgen eine lauffähige version jedes produkts. Diese im grunde genommen sehr aufwendige methode der täglichen synchronisation hat den vorteil, dass das projektteam wie auch die firmenleitung täglich über den projektfortschritt informiert bleibt. Wir zitieren aus Cusumano und Selby (1996, s. 207):

> *Wir bauen jeden Tag, und zwar genau um 17 Uhr, eine Zwischenversion [...] selbst wenn am nächsten Tag Feiertag ist, und wir genau wissen, dass niemand die Zwischenversion benötigt. Alle Beteiligten bekommen ein Gefühl für den Prozess und sehen, daß das Projekt kontrolliert abläuft.*

bzw. (S. 208)

> *Für uns ist wichtig, dass alle Beteiligten vor Feierabend synchronisieren, daß also jeder Entwickler seine Änderungen abgleicht. Wenn man dies aufschiebt, bekommt man irgendwann einmal größere Konflikte beim Zusammenmischen.*

Programmierer sind sehr daran interessiert, einen code einzustellen, der den bau der zwischenversion nicht behindert, denn personen, die einen abbruch beim bau der zwischenversion verursachen, werden auf die eine oder andere art und weise bestraft, z.b. dadurch dass sie die wenig geliebte aufgabe des *build master* übernehmen müssen; je nach abteilung wird auch eine geldstrafe fällig (5–6 $) oder es wird zu drastischeren mitteln gegriffen, zitat (Cusumano und Selby 1996, S. 208):

> *Wir hatten damals Ziegenhörner. Wer einen Abbruch verursachte, mußte die Ziegenhörner aufsetzen und so lange aufbehalten, bis ein anderer den Zwischenversionsbau zum Absturz brachte.*

Jeden morgen wird die erzeugte zwischenversion von einem *usability lab* übernommen, die sozusagen die rolle der anwender simulieren und das produkt täglich auf herz und nieren prüfen. Bei Microsoft steht zahlenmässig jedem

programmierer ein tester gegenüber, ein beweis für das gewicht, das die firma auf qualität legt.

Zu bestimmten zeitpunkten (meilensteine) findet ausserdem nochmals eine separate stabilisierung der quellprogramme statt in dem sinne, dass dann keine neuen vorrichtungen mehr in den code eingeführt werden, sondern lediglich fehler beseitigt werden.

Cusumano (2007b) macht sich aus betriebswirtschaftlicher sicht gedanken um die strategie von Microsoft für die nächsten jahre. Dabei macht er sich vor allem sorgen um das wachsende alter der Microsoft-firmenführung:

> *Can a veteran leadership team envision the future of the changing software business before it happens and make the foresighted kinds of investments that have brought so much recent attention to Google, Apple, and Web 2.0 entrants such as YouTube? [...] Figuring out how to build the next version of Windows the operating system, however difficult that may be, seems trivial in comparison to building the next version of Microsoft the company.*

Kritisiert werden von Cusumano allerdings in gleichem atemzug die hohen ausgaben für forschung und entwicklung: 2006 hat Microsoft 15% seines Einkommens (6,6 milliarden Dollar) dafür ausgegeben, ohne dass dadurch entsprechende innovationen in den produkten zu erkennen seien. Microsoft müsse lernen, sein geld gezielter auszugeben, statt es mit der giesskanne über alle stellen zu verteilen, die auch nur einigermassen erfolgversprechend seien.

Den augenblicklichen trend in der softwarebranche, die eigentliche software regelrecht zu verschenken und dann von erweiterten leistungen (service, beratung) zu leben, wie er von IBM und SAP vorgemacht wird, hat Microsoft noch nicht für sich entdeckt; MSN/Windows Live und Office Live kopieren immerhin das von Google, Yahoo und AOL vorgemachte geschäftsmodell, „kostenlose“ leistungen mit indirekten einkünften (z. b. durch werbung) zu kombinieren.

Sparte	*Umsatz*	*Gewinn*	*proz. gewinn*
Windows	13,2 Mrd.	10,2 Mrd.	77%
Office	11,8 Mrd.	8,2 Mrd.	69%
Server und werkzeuge	11,5 Mrd.	4,3 Mrd.	37%
Unternehmenslösungen	919 Mrd.	24 Mio.	2,6 tausendstel %
MSN	2,3 Mrd.	-77 Mio.	-3,3%
Handy, PDA etc.	377 Mio.	2 Mio.	0,5%
XBOX, Spiele	4,3 Mrd.	-1,3 Mrd.	-30%

Tabelle 11: Microsoft gewinnanalyse (USD)

Interessant sind auch die zusammenstellungen von zahlen aus dem geschäftsbericht von 2006, den Cusumano analysiert hat. Wie nicht anders zu erwarten, macht Microsoft sein bestes Geld mit Windows und Office (s. tab. 11).

Bereich	*Umsatz in prozent*
OEM	34
Unternehmen	41
Privatkunden	27

Tabelle 12: Microsoft kundenkreise

Auf der Kundenseite ist es (tab. 12) für Microsoft wichtig, die OEM-kunden und geschäftskunden besonders gut zu behandeln, diese machen zusammen reichlich zwei drittel des umsatzes. (Ein kleiner schönheitsfehler in dieser tabelle ist es, dass sich die zahlen zu 102% addieren, nach Auskunft von Prof. Cusumano eine Auswirkung von Rundungen.)

7.1 Vergleich mit XP

Da sowohl die Microsoft-methode als auch der XP-prozess letztlich in der hackerkultur gründen, liegt es nahe, diese beiden ansätze zu vergleichen. Cusumano (2007a) hat dies getan und kommt zu den folgenden übereinstimmungen (die er selbst als teilweise oberflächlich kennzeichnet) bzw. unterschieden:

Planung mit user stories Wie bei den user stories von XP stehen auch bei Microsoft kleine stücke von funktionalität, die *features*, im vordergrund der betrachtung.

Kleine releases Beim XP-prozess bekommt der kunde sein produkt „häppchenweise“ in kleinen inkrementen; eine analogie im haus Microsoft (das einen „auftraggeber“ im klassischen sinne nicht kennt) besteht hier in den *daily builds*, die dem *usability lab* als simuliertem auftraggeber in kurzen zyklen übergeben werden.

System-metapher Die entsprechung zum im XP-prozess angestrebten gemeinsamen sicht des systems sind im haus Microsoft die *vision statements*, die in kürze beschreiben, welche schlüssel-*features* im system enthalten bzw. nicht enthalten sein sollen.

Einfacher entwurf O-Ton Cusumano: *„There is no Microsoft-style analogy for this point.“* In der Tat rät er der firma Microsoft dringend, diesem punkt stärkere beachtung zu schenken (Cusumano 2007b).

Stetiges testen Die idee des *test driven design* ist bei Microsoft nicht präsent, aber es wird trotzdem grosser wert auf tägliche tests gelegt. Es sind auch (in der entwicklung des Internet Explorer und des NetMeeting) fälle dokumentiert, in denen die team-manager sich statt der vorgabe von programmierentscheidungen auf das schreiben von testfällen verlegt haben.

Refactoring findet bei Microsoft selbstverständlich ebenfalls statt, auch wenn es nicht explizite vorschrift im prozess ist.

Pair programming Auf den ersten blick reduziert *pair programming* die programmierleistung eines teams auf die hälfte; deshalb muss man sich nicht wundern, dass es bei einer firma, die auf kommerziellen erfolg ausgerichtet ist, in dieser form nicht vorgesehen ist. Allerdings gibt es bei Microsoft in der regel für jeden programmierer einen tester. Cusumano glaubt, dass sich mit dem XP-ansatz des *pair programming* gegenüber der Microsoft-methode letztlich geld sparen liesse. In der Tat berichtet der Spiegel (Dworschak 2009), dass Steven Sinofsky als leiter der entwicklung von Windows 7 das *pair programming* bei Microsoft eingeführt habe.

Collective code ownership Microsoft schätzt es – wie viele industrieunternehmen – für jede komponente einen (haupt-) verantwortlichen zu haben und widersetzt sich daher der XP-idee, dass jeder jedes stück code ändern darf. Da in langlebigen software-produkten aber das eigentum an code-elementen über die zeit wechselt, kommt es zuletzt doch zu einer abgemilderten form der *collective code ownership*.

Stetige integration Eine gewisse analogie zur XP-methodik findet sich in dem *synch and stabilize*-ansatz wieder, auch wenn die intensität und effektivität der stetigen integration bei Microsoft vielleicht nicht so hoch ist wie im XP-prozess.

Minimierung von überstunden Zitat: „Das XP ideal nach Beck ist gegenseitiger respekt und zusammenarbeit zwischen programmierern. Im gegensatz dazu sehen wir es bei Microsoft und den meisten anderen softwareunternehmen häufig, dass marketing-ziele in projekten vorrang vor professionellen zielen bekommen, was firmen dazu zwingt, überstunden als kompensation für unrealistische pläne oder schlechtes projektmanagement und teamwork einzusetzen. Ein anderer ansatz, der bei Windows und anderen Microsoft-gruppen üblich ist, anstatt sich einfach auf überstunden zu verlassen, ist es, produkte verspätet herauszugeben, manchmal um jahre."

Kunde vor ort Da es für Microsoft-produkte einen auftraggeber im herkömmlichen sinne nicht gibt, entfällt dieser punkt. Produkt- und programmmanager versuchen sich aber in die lage potentieller kunden zu versetzen.

Programmiernormen Im ziel, die gleichen entwurfs-notationen oder -normen zu verwenden, gibt es kaum unterschiede zwischen der XP-methode und Microsoft.

In summa schreibt Cusumano über beide ansätze: „Ich glaube, dass interative entwurfspraktiken für jedes softwareprojekt nützlich sind, bei dem die vorgegebenen spezifikationen unvollständig sind – was praktisch immer der fall ist.“

Kontrollfragen

1. Was ist bei Microsoft eine *feature group*? Was ist ihre zuständigkeit, wieviele personen umfasst sie und wie ist sie zusammengesetzt?
2. Welche vorteile bieten die *feature groups* für die organisation der firma?
3. Beschreiben sie die *sync-and-stabilise*-methode!

8 Leistungsverbesserung von software

The most effective way to make a program faster is to use a better algorithm. (Kernighan und Pike 1999)

Dataless suppositions are like suppositories: they provide only temporary relief and they should be shoved in the same place. (G. V. Neville-Neil, kode vicious, ACM Queue Vol. 7 No. 5, Nov./Dec. 2007)

Many companies run their servers with the fastest processors and the maximum amount of memory. Usually companies do this because they have hired the wrong people to write their software. (Neville-Neil 2008b)

Leistungsverbesserung setzt zunächst einmal leistungsmessung voraus. Schon Knuth (1971) wies nach, dass weniger als 4 prozent eines programms mehr als 50 % der laufzeit verbrauchen[34]. Bentley (1988) sagt: „Ich habe dieses klassische papier in den letzten zehn jahren mindestens einmal im jahr gelesen und es wird jedesmal besser. Ich empfehle es dringend.“ Die zahlen von Knuth legen den schluss nahe, dass ein „optimieren“ am falschen platz eine grosse verschwendung von ressourcen sein kann.

Kernighan und Pike (1999) sagen dazu:

> *Vor langer zeit unternahmen programmierer grosse anstrengungen, ihre programme effizient zu machen, weil computer langsam und teuer waren. Heute sind die maschinen viel billiger und schneller, also ist die notwendigkeit für absolute effizienz viel geringer. Ist es immer noch lohnenswert, sich über leistung gedanken zu machen?*
>
> *Ja, aber nur wenn das problem wichtig ist, das programm wirklich zu langsam ist und wenn man erwarten kann, dass es in einer weise beschleunigt werden kann, dass korrektheit, robustheit und klarheit erhalten bleiben. Ein schnelles programm, das die falsche antwort liefert, spart keine zeit.*
>
> *Die erste regel der optimierung heisst deshalb:* tu's nicht.

Sie beschreiben anschliessend die beste strategie zur leistungssteigerung von programmen, wenn sie denn sinnvoll und nötig ist, wie folgt:

1. Benutze die einfachsten, saubersten algorithmen und datenstrukturen, die für die aufgabe geeignet sind.

2. Miss die leistung, um zu sehen, ob änderungen nötig sind.

3. Schalte compileroptionen ein, um den schnellstmöglichen code zu erzeugen.

[34]Dies gilt unter der einschränkung des von Knuth untersuchten programm-materials aus dem umfeld der numerischen verfahren.

4. Stelle fest, welche änderungen am programm die grösste wirkung haben werden.
5. Führe die änderungen jeweils einzeln durch und miss jeweils erneut.
6. Hebe alle versionen auf, um revisionen dagegen zu testen.

Sie stellen überdies fest:

> *„Messungen sind ein unverzichtbarer teil der leistungsverbesserung, denn argumentation und intuition sind trügerische führer und müssen durch werkzeuge wie zeitmessungskommandos und* profiler *unterstützt werden. Leistungsverbesserung hat viel gemeinsam mit testen, inklusive solcher techniken wie automatisierung, sorgfältige buchführung und verwendung von regressionstests, die sicherstellen, dass änderungen die korrektheit erhalten und nicht vorherige verbesserungen rückgängig machen."*

Zum messen von programmausführungen verwenden wir sogenannte *„profiler"*. Abseits von leistungsdaten können solche *profiler* auch zu ganz rudimentären korrektheitsüberlegungen oder zur gewinnung ganz anderer einsichten über das programm eingesetzt werden.

Eine erste grobe unterscheidung der *profiler* trifft man wie folgt:

Line count profiler: Zählen, wie oft eine zeile ausgeführt wurde (auch nützlich für elementare konsistenzprüfungen oder pfadüberdeckung)

Run time profiler: Zählen, wie oft eine prozedur aufgerufen wurde und wieviel zeit dabei jeweils verbracht wurde.

In der regel läuft die messung in beiden fällen so ab:

1. Eine spezifische compileroption reichert den code um messinstruktionen an.
2. Während des programmlaufs wird messinformation in eine datei geschrieben.
3. Nach dem programmlauf wird die messinformation durch ein separates programm aufbereitet.

Einfache *run time profiler* arbeiten zuweilen auch so, dass in bestimmten intervallen das programm unterbrochen wird und der wert des befehlszeigerregisters anhand einer tabelle einer prozedur des programms zugeordnet wird. Damit sind allerdings nur ganz grobe statistische aussagen möglich; der

vorteil ist, dass das eigentliche programm gar nicht modifiziert zu werden braucht.

Am beispiel eines primzahlprogramms führen wir eine codeverbesserung vor; dieses beispiel stammt von Bentley (1988, kap. 1). Das ausgangsprogramm lautet:

```
            int prime (int n)
            {   int i;
999             for (i = 2; i < n; i++)
78022               if  (n % i == 0)
831                     return 0;
168             return 1;
            }

            main( )
            {       int i, n;
1                   n = 1000;
1                   for  (i = 2;  i <= n; i++)
999                     if  (prime(i))
168                         printf("%d\n", i);
            }
```

Die zahlen am linken zeilenrand wurden hier durch einen *line count profiler* erzeugt. Wir können so z.b. erkennen, dass 78022 mal auf teilbarkeit geprüft wurde.

Das kann man drastisch reduzieren, wenn man die obergrenze für den teilbarkeitstest heruntersetzt. Das hätte man bei sorgfältiger überlegung schon von vorneherein getan, aber hier geht es ja darum, die verbesserungen vorzuführen. Das verbesserte programm zusammen mit den *line count profiles* sieht dann so aus:

```
            int root(int n)
5456        { return (int) sqrt((float) n); }

            int prime(int n)
            {   int i;
999             for (i = 2; i <= root(n); i++)
5288                if (n % i == 0)
831                     return 0;
168             return 1;
            }

            main( )
            {       int i, n;
1                   n = 1000;
1                   for (i = 2; i <= n; i++)
999                     if (prime(i))
168                         printf("%d\n", i);
            }
```

Wir sehen, dass die teilbarkeitstests von 78022 auf 5288 zurückgegangen sind. Glücklicherweise hat sich an der zahl der gefundenen primzahlen (168) nichts geändert. Bentley bemerkt überrascht, dass die „verbesserte“ version trotzdem auf seiner VAX deutlich langsamer läuft als das ausgangsprogramm. Um zu sehen, wohin die zeit geht, verwenden wir einen *run time profiler*; in der originalarbeit von Bentley genügt es dazu, wie in den beispielen aufgezeigt, die primzahlen bis zu 1000 ausrechnen zu lassen. Die von Bentley verwendete VAX kann sich aber mit heutigen rechnern überhaupt nicht messen; wir müssen hier schon bis 100.000 gehen, um signifikante zahlen zu erhalten. In einem „vanilla Unix“ gibt es für die zeitmessung ein programm `prof`, das wie folgt bedient wird:

```
cc -lm -p -o primes primes.c   # Mit mathe-bibliothek und profiling
primes                         # Programmlauf, erzeugt datei mon.out
prof primes                    # Auswertung und aufbereitung
```

Verbesserte informationen bekommt man mit dem *call graph profiler* `gprof` von Susan Graham u. a. (1982); dort wird für jede funktion separat die zeit ausgewiesen, die auf aufrufe von jeder anderen funktion aus aufgewendet wurde. Das heisst also, statt einer information „funktion f hat insgesamt x sekunden laufzeit beansprucht“, kann man hier erfahren „funktion f hat y sekunden laufzeit für aufrufe aus einer funktion g verbraucht und z sekunden für aufrufe aus einer funktion h“.

Die im folgenden aufgeführten zahlen sind historisch; sie entstanden auf einer IBM RiscSystem/6000. Hier zunächst die ergebnisse für das ausgangsprogramm:

```
Name                 %Time     Seconds     Cumsecs  #Calls   msec/call
.prime                99.9      401.16      401.16   99999      4.0116
._doprnt               0.0        0.16      401.32    9592      0.017
.__mcount              0.0        0.10      401.42
.__mcount              0.0        0.07      401.49
.main                  0.0        0.07      401.56       1     70.
.printf                0.0        0.03      401.59    9592      0.003
.strchr                0.0        0.03      401.62
.strlen                0.0        0.02      401.64
.printf.GL             0.0        0.02      401.66
.exit                  0.0        0.00      401.66       1      0.
```

Diese liste enthält in der „Name“-spalte die namen von prozeduren (C-funktionen); funktionen bilden hier die sogenannte *messgranularität*. Wir erkennen unsere eigene funktion `prime` in der ersten und das hauptprogramm `main` in der fünften zeile. Die restlichen funktionen sind bibliotheksfunktionen, die teilweise *inline* compiliert wurden (`printf`, `exit`) und teilweise vom

linker eingebunden wurden. Die liste wurde am ende abgekürzt; hier kommen in wirklichkeit noch 17 weitere zeilen. Die spalte „%Time“ enthält den anteil der gesamten laufzeit, der auf die entsprechende funktion entfiel und „Seconds“ die hier insgesamt verbrachten CPU-sekunden. Die zeilen in der tabelle sind nach absteigenden CPU-sekunden geordnet, so dass die „grossverbraucher“ am anfang der liste besonders ins auge fallen. In der spalte „Cumsecs“ („cumulative seconds“) sind zur vermeidung besonderer rechnungen die „Seconds“-werte bis zur gegenwärtigen zeile aufsummiert. Wir erkennen, dass dieser programmlauf insgesamt 401,66 sekunden gedauert hat, also reichlich sechseinhalb minuten. Zwei weitere spalten („#Calls“ und „msec/call“) geben die anzahl der aufrufe und die durchschnittlichen millisekunden pro einzelnem aufruf an. Wir sehen, dass die 99.999 aufrufe der funktion `prime` durchschnittlich rund 4 millisekunden verbraucht haben und dass wir insgesamt 9.592 primzahlen gefunden haben. Für die vom linker eingebundenen bibliotheksfunktionen erhalten wir diese angaben nicht, da die bibliotheken aus effizienzgründen ohne messoptionen übersetzt wurden. Bei manchen programmierer-freundlichen Unixen kann man durch eine linker-option alternative bibliotheken mit messfunktionen einbinden lassen.

Nun die versprochenen zahlen für das modifizierte programm:

```
Name                 %Time     Seconds     Cumsecs  #Calls   msec/call
.root                 28.3        3.38        3.38 2755286      0.0012
.prime                21.4        2.55        5.93   99999      0.0255
.sqrt                 18.7        2.23        8.16
.__mcount             17.0        2.03       10.19
.itrunc               12.5        1.49       11.68
.__mcount              0.7        0.08       11.76
._doprnt               0.4        0.05       11.81    9592      0.005
.mcount                0.4        0.05       11.86
.strchr                0.3        0.03       11.89
.main                  0.2        0.02       11.91       1     20.
.printf                0.1        0.01       11.92    9592      0.001
.strlen                0.1        0.01       11.93
.__mcount_construct_   0.1        0.01       11.94
.exit                  0.0        0.00       11.94       1     0.
```

Man sieht, dass die meiste zeit in dem unterprogramm für die wurzelfunktion verbracht wird, welches fast drei millionen mal aufgerufen wird. Das ist auch der grund, warum Bentleys programm so langsam lief, denn auf der VAX wird typischerweise ein unterprogramm zur näherungsweisen berechnung der quadratwurzel verwendet, was dementsprechend langsam ist. Auf einer IBM RS/6000, die über einen schnellen coprozessor verfügt, hat sich das programm um den faktor 33 beschleunigt; wir brauchen jetzt nur noch knapp 12 sekunden.

Unser fehler ist natürlich, dass wir die wurzelfunktion in der innersten schleife aufrufen. Das stellen wir jetzt ab:

```
int prime(int n)
{    int i, bound;
     bound = root (n);
     for (i = 2; i <= bound; i++)
          if (n % i == 0)
               return 0;
     return 1;
}
```

Damit ergibt sich:

Name	%Time	Seconds	Cumsecs	#Calls	msec/call
.prime	70.4	2.59	2.59	99999	0.0259
.__mcount	5.7	0.21	2.80		
._doprnt	4.3	0.16	2.96	9592	0.017
.root	3.0	0.11	3.07	99999	0.0011
.fwrite	2.7	0.10	3.17	19184	0.0052
.sqrt	2.4	0.09	3.26		
.__mcount	2.4	0.09	3.35		
._xwrite	1.6	0.06	3.41	9592	0.006
.itrunc	1.6	0.06	3.47		
._xflsbuf	1.6	0.06	3.53	9592	0.006
.main	1.1	0.04	3.57	1	40.
.memchr	0.8	0.03	3.60	19184	0.0016
.strlen	0.8	0.03	3.63		
.memcpy	0.5	0.02	3.65		
.printf.GL	0.3	0.01	3.66		
.write	0.3	0.01	3.67	9592	0.001
.printf	0.3	0.01	3.68	9592	0.001
.exit	0.0	0.00	3.68	1	0.

Wir kommen also jetzt schon unter 4 sekunden und finden trotzdem immer noch 9.592 primzahlen. Als weitere verbesserung nehmen wir die ersten drei fälle (teilbarkeit durch 2, 3 und 5) aus der schleife heraus. Diese fälle treten so häufig auf, dass es sich hierfür lohnen sollte, die schleifeninitialisierung einzusparen:

```
int root(int n)
{    return (int) sqrt((float) n); }

int prime(int n)
{    int i, bound;
     if (n % 2 == 0)
          return (n == 2);
     if (n % 3 == 0)
          return (n == 3);
     if (n % 5 == 0)
```

```
            return (n == 5);
        bound = root(n);
        for (i = 7; i <= bound; i = i+2)
            if (n % i == 0)
                return 0;
        return 1;
        }

    main( )
    {       int i, n;
            n = 1000;
            for (i = 2; i <= n; i++)
                if (prime(i))
                    printf("%d\n", i):
    }
```

Wie man sieht, ist diese einsparung nicht mehr so wesentlich, wir bekommen nur noch eine anderthalbfache beschleunigung:

Name	%Time	Seconds	Cumsecs	#Calls	msec/call
.prime	58.8	1.37	1.37	99999	0.0137
._doprnt	6.0	0.14	1.51	9592	0.015
.main	6.0	0.14	1.65	1	140.
.fwrite	5.6	0.13	1.78	19184	0.0068
.__mcount	5.2	0.12	1.90		
.strchr	2.6	0.06	1.96		
.root	2.6	0.06	2.02	26665	0.0023
.__mcount	2.6	0.06	2.08		
.itrunc	2.1	0.05	2.13		
._xwrite	2.1	0.05	2.18	9592	0.005
.sqrt	1.7	0.04	2.22		
.memchr	1.3	0.03	2.25	19184	0.0016
.write	1.3	0.03	2.28	9592	0.003
.printf.GL	0.4	0.01	2.29		
.mcount	0.4	0.01	2.30		
._xflsbuf	0.4	0.01	2.31	9592	0.001
.memcpy	0.4	0.01	2.32		
.printf	0.4	0.01	2.33	9592	0.001
.exit	0.0	0.00	2.33	1	0.

Die wurzelfunktion auf der VAX ist so langsam, dass es sich immer noch lohnt, sie ganz zu vermeiden:

```
    for (i = 7; i*i <= n; i = i+2)
        if (n % i == 0)
            return 0;
    return 1;
```

Für die RS/6000 ergibt sich mit dieser Änderung

```
Name                %Time     Seconds     Cumsecs  #Calls   msec/call
.prime               84.8        2.46        2.46   99999      0.0246
._doprnt              5.2        0.15        2.61    9592      0.016
.__mcount             4.8        0.14        2.75
.main                 2.8        0.08        2.83       1     80.
.printf               1.7        0.05        2.88    9592      0.005
.__mcount             0.7        0.02        2.90
.exit                 0.0        0.00        2.90       1      0.
```

d. h. wir haben hier überoptimiert: Die laufzeit ist wieder länger geworden. Auf anderen rechnern kann das ergebnis dieser „optimierung“smassnahme ganz anders aussehen. Zum vergleich habe ich ein MacBook Air mit MacOSX Snow Leopard herangezogen.

Damit kommen wir insgesamt zu folgendem überblick:

Programm	n = 10.000 VAX	n = 100.000 VAX	n = 100.000 RS/6000	n = 100.000 MacBook Air
P1. Einfache version	169	?	401.66	2.172
P2. Grenze bei $\sqrt{n}$	124	2850	11.94	0.020
P3. Wurzel ausserhalb schleife	15	192	3.68	0.018
P4. Spezialfälle 2, 3, 5	5.7	78	2.33	0.014
P5. Multiplikation statt wurzel	3.5	64	2.9	0.016

Auch bei der verwendung eines *profiler* können einem immer noch irrtümer unterlaufen. Bentley (1988) berichtet in

> Profiling *zeigte, dass die hälfte der laufzeit eines betriebssystems in einer schleife mit nur wenigen instruktionen verbracht wurde. Umschreiben dieser schleife in mikrocode machte sie um eine grössenordnung schneller, aber verbesserte die systemleistung überhaupt nicht: Die optimierungsgruppe hatte die leerlaufschleife des systems optimiert!*

Ein anderes schönes beispiel zur leistungsverbesserung zeigen Kernighan und Pike (1999); hier geht es um einen *spam filter*, der e-mails zurückweisen soll, die bestimmte textmuster enthalten. Ein erster ansatz für eine funktion, die eine nachricht `mesg` nach vorkommen von mustern `pat[i]` durchsuchen soll, sieht deshalb wie folgt aus:

```
/* isspam: test mesg for occurrence of any pat */
int isspam (char *mesg)
{
   int i;

   for (i=0; i < npat; i++)
      if (strstr(mesg, pat[i]) != NULL) {
         printf("spam: match for '%s'\n", pat[i]);
         return 1;
      }
   return 0;
}
```

Diese lösung stellte sich im betrieb als viel zu langsam heraus, obwohl man davon ausgehen kann, dass funktionen der C-library wie `strstr` hochoptimiert sind. Um zu verstehen, wieso die obige lösung trotzdem langsam ist, muss man in die C-library hineinschauen. Leicht vereinfacht, stellt sich die implementierung von `strstr` wie folgt dar:

```
/* simple strstr: use strchr to look for first character */
char *strstr(const char *s1, const char *s2)
{
   int n;

   n = strlen(s2);
   for (;;) {
      s1 = strchr(s1, s2[0]);
      if (s1 == NULL)
         return NULL;
      if (strncmp(s1, s2, n) == 0)
         return (char *) s1;
      s1++;
   }
}
```

Aus effizienzgründen sucht `strstr`, dessen aufgabe es ist, eine zeichenkette innerhalb einer anderen zeichenkette zu suchen, zunächst nach dem ersten buchstaben der gesuchten Zeichenkette. Damit können grosse bereiche der zu durchsuchenden zeichenkette gleich übersprungen werden. Erst wenn der erste buchstabe gefunden ist, wird mit `strncmp` der rest verglichen. Für den hier vorliegenden anwendungszweck stellt es sich als ungünstig heraus, dass `strncmp` die länge der gesuchten zeichenkette als argument braucht und dass `srtstr` keine andere möglichkeit hat, als diese länge ausdrücklich zu berechnen. In wirklichkeit ist die länge jedes einzelnen suchmusters aber im voraus bekannt. Ausserdem stellen sich nun einige nachteile der C *string library* heraus: Durch die entscheidung, dass zeichenketten durch ein nullbyte beendet werden, muss sowohl `strchr` jedes einzelne zeichen ausser mit dem suchzeichen auch noch mit dem nullbyte vergleichen; `strncmp` muss sogar in beiden zeichenketten auf das nullbyte prüfen, bevor zwei zeichen miteinander verglichen werden. Dazu kommt der verwaltungsaufwand für das aufrufen der funktionen `strlen`, `strchr` und `strncmp`.

Als ersten ansatz zur beschleunigung des *spam filters* wurde deshalb eine auf diesen anwendungsfall spezialisierte version von `strstr` geschrieben, die keine anderen funktionen mehr aufrief. Leider konnte dadurch das programm um nur 30% beschleunigt werden, was nicht ausreichte.

An dieser stelle wird es zeit, sich zu überlegen, ob wir das richtige problem lösen. Bis jetzt sind wir von der vorstellung ausgegangen, *eine* zeichenkette („suchmuster“) in einer anderen zeichenkette („nachricht“) zu suchen.

Diese vorstellung beinhaltet, dass wir für jedes der suchmuster im wesentlichen doch die ganze nachricht durchforsten. Das bedeutet zeichenvergleiche in der grössenordnung `strlen(mesg)`×`npat`. Wir lösen also das falsche problem, denn in wirklichkeit suchen wir nach einer zeichenkette aus einer bestimmten menge innerhalb einer anderen zeichenkette. Sinnvoller wäre deshalb, die nachricht nur einmal zu durchsuchen und dabei nach allen mustern zu suchen:

```
for (j = 0; mesg[j] != '\0'; j++)
    if (some pattern matches starting at mesg[j])
        return 1;
```

Der leistungsgewinn kommt jetzt daher, dass wir gar nicht alle muster anschauen müssen: wenn wir bei `mesg[j]` stehen, kommen sowieso nur solche muster in frage, die mit `mesg[j]` beginnen. Theoretisch könnte dies einen beschleunigungsfaktor von 52 bedeuten (26 gross- + 26 kleinbuchstaben), aber die buchstaben kommen nicht gleichverteilt in einer nachricht vor. Mit dem folgenden programm, das eine reihe von vorberechneten tabellen verwendet, kommen wir zu einer beschleunigung von 7 bis 15 gegenüber dem ursprünglichen programm:

```
int patlen[NPAT];                     /* length of pattern */
int starting[UCHAR_MAX+1][NSTART];    /* patterns starting with char */
int nstarting[UCHAR_MAX+1];           /* number of such patterns */

/* isspam_ test mesg for occurrence of any pat */
int isspam(char *mesg)
{
    int i, j, k;

    unsigned char c;

    for (j = 0; (c = mesg[j]) != '\0'; j++) {
        for (i = 0; i < nstarting[c]; i++) {
            k = starting[c][i];
            if (memcmp(mesg+j, pat[k], patlen[k]) == 0) {
                printf("spam: match for '%s'\n", pat[k]);
                return 1;
            }
        }
    }
    return 0;
}
```

8.1 Fallstricke

Millsap (2010) weist auf einige fallstricke hin, in die man bei der leistungsverbesserung leicht geraten kann:

- Verwechslung von leistung mit durchsatz: Angenommen, ein system kann pro sekunde 1.000 aufgaben einer bestimmten art erledigen. Dann kann nicht angenommen werden, dass die durchschnittliche antwortzeit eine tausendstel sekunde ist, denn der durchsatz könnte durch parallelsierung erreicht worden sein. (Im schlimmsten fall standen 1.000 prozessoren zur verfügung, und die antwortzeit ist eine sekunde.)
- Perzentil-spezifikationen: Eine durchschnittliche antwortzeit von einer sekunde sagt überhaupt nichts aus, wenn antwortzeiten stark streuen. In so einem fall wäre es besser zu wissen, dass etwa das 90. perzentil eine sekunde ist; dann werden 90% der anfragen in einer sekunde oder schneller erledigt.
- Amdahls gesetz: die beschleunigung, die durch eine leistungsverbesserung erreicht werden kann, ist proportional zu dem prozentsatz, in dem ein system die beschleunigte funktion verwendet. (Daher auch die empfehlung, vor und nach der verbesserung zu messen!) Wenn ausserdem die kosten einer verbesserungsmassnahme herangezogen werden – was sich stets empfiehlt – kann es besser sein, eine billigere massnahme mit weniger wirkung zu bevorzugen gegenüber einer sehr teueren massnahme mit grösserer wirkung.

Kontrollfragen

1. Welches ist die beste methode, die laufzeit eines programms zu verbessern?
2. Wieso soll man über ein programm zuerst detaillierte leistungsdaten messen, bevor man optimierungen durchführt?
3. Was leisten profiler? Welche arten davon gibt es?

9 Softwarewerkzeuge

Ein softwareprozess kann normalerweise nicht sinnvoll ohne eine reihe von werkzeugen durchgeführt werden. Am oberen ende der werkzeugkette finden sich sogenannte IDEs („Integrated Development Environments"), welche den gesamten softwareprozess unter einer einheitlichen oberfläche kapseln und damit glauben, etwas gutes zu tun. Diese denkweise findet sich auch in kommerziell erhältlichen programmpaketen (Word, Excel, ...) wieder. Merkwürdigerweise sind die meisten profis wenig begeistert von solchen ansätzen, einerseits deswegen, weil solche softwarepakete auf diese weise unnötig komplex werden (s. abb. 51) und von ihren zahlreichen zwecken nur wenige wirklich vollkommen erfüllen, aber hauptsächlich deshalb, weil solche komplexen werkzeuge häufig nach der devise „take it or leave it" konstruiert sind, d. h. kaum raum für eigene anpassung oder gar auswechslung unvollkommener komponenten gegen fremde, bessere lassen. Eine ausnahme in dieser richtung bietet z. B. *Eclipse* (`www.eclipse.org`), das sich durch *plugins* in jeder richtung erweitern lässt. Die komplexität des werkzeugs steigt dadurch allerdings weiter an; bedienung und beratung („hotline") werden komplizierter, weil jeweils auch darauf geachtet werden muss, ob die richtigen plugins installiert wurden.



Abbildung 51: Das universalwerkzeug

Eine radikal andere einstellung findet sich in dem betriebssystem Unix, das in seinem diversen varianten nicht zuletzt deshalb bei programmentwicklern so beliebt ist, weil es eine vielzahl nützlicher kleiner werkzeuge anbietet, die

sich einzeln oder in kombination wunderschön verwenden lassen, um im softwareprozess sinnvolle aktionen automatisch ablaufen zu lassen. Damit dies funktionieren kann, sind zwei voraussetzungen massgeblich:

1. Mit wenigen (offensichtlichen) ausnahmen operieren alle Unix-werkzeuge auf textdateien, die unter anderem auch mit standard-editoren bearbeitet werden können.

2. Über den mechanismus der *pipe* kann die ausgabe des einen programms sofort als eingabe des nächsten verwendet werden, ohne dass die programme auf diese art der kooperation eigens eingerichtet sein müssen.

Wenn es sich herausstellt, dass eine bestimmte aufgabe nicht von den Unix-werkzeugen erledigt werden kann, ist es häufig sehr einfach, ein relatives kleines und übersichtliches programm zu schreiben, das sich über den mechanismus der *pipe* zu einer werkzeugkette kombinieren lässt, welche das vorliegende problem löst.

Übrigens sei am rande auch bemerkt, dass die werkzeuge von Unix hauptsächlich deshalb so gut sind, weil sie auf soliden und universell einsetzbaren konzepten aus der theoretischen informatik basieren. Eine intensive beschäftigung mit der theorie ist deshalb — im gegensatz zur landläufigen meinung vieler studierender — essentiell für einen erfolg in der praxis! Eine sehr gute einführung in die Unix-ideologie mit vielen interessanten beispielen findet sich bei Raymond (2003). Ein kleines, übersichtliches beispiel schildert Bentley (1986): Angenommen, wir wollen wissen, welches die 15 häufigsten wörter in einem auf viele dateien verteilten textdokument sind und wie häufig jedes dieser wörter vorkommt. Der Unix-programmierer würde diese aufgabe sehr schnell mit einer pipe erledigen:

```
cat $*|tr -cs A-Za-z '\012'|tr A-Z a-z|sort|uniq -c|sort -r -n|sed 15q
```

Die erklärung dafür folgt[35]:

1. `cat $*` verkettet alle dateien in eine einzige,
2. `tr -cs A-Za-z '\012'` verwandelt alle folgen (`-s` wie single) von zeichen, die keine buchstaben sind (`-c` wie complement), in je einen einzigen line feed,
3. `tr A-Z a-z` verwandelt alle grossbuchstaben in kleinbuchstaben,
4. `sort` sortiert diese zeilen *alphabetisch*,
5. `uniq -c` eliminiert aufeinanderfolgende identische zeilen und fügt am ende die zahl gefundener gleicher zeilen an,

[35]Bentley geht wie alle Amerikaner davon aus, dass texte in US-ASCII geschrieben werden. Will man auch mit umlauten umgehen können, würde man in den `tr`-anweisungen jeweils statt `A-Z` und `a-z` die wendungen `'[:upper:]'` und `'[:lower:]'` verwenden.

6. `sort -r -n` sortiert diese zeilen in absteigender (`-r` wie reverse) *numerischer* (`-n`) folge und
7. `sed 15q` listet die ersten 15 zeilen der ergebnisdatei.

Bezogen auf das vorliegende skriptum ergibt sich übrigens hieraus

```
1721 die
1499 der
 965 in
 949 und
 758 von
 625 ist
 594 zu
 576 das
 495 werden
 473 eine
 418 dass
 415 es
 402 den
 400 ein
 370 auf
```

(Wie kaum anders zu erwarten, sind die häufigsten wörter nicht gerade die am meisten bedeutungstragenden.)

Ich stelle hier anstelle von IDEs lieber individuelle softwarewerkzeuge vor, weil sie einfacher zu beschreiben sind als komplexe werkzeugkästen und weil man hier noch besser erkennen kann, wie die mechanismen funktionieren. Letztlich kann man die IDEs nicht wirklich verstehen, wenn einem die grundlegenden mechanismen nicht klar sind.

9.1 awk, grep, sed

Drei werkzeuge, die zusammen mit dem programmierbaren kommandointerpreter („shell“) unglaubliches leisten können, sind `awk`, `grep` und `sed`. Von diesen ist den studierenden meist nur `grep` wirklich bekannt:

sed ist der *stream editor*, er operiert im gegensatz zu den „normalen“ editoren auf einem *zeichenstrom*, das heisst, er liest eine eingabedatei zeichen für zeichen und schreibt dabei gleichzeitig eine ausgabedatei. Deswegen gibt es bei ihm keinerlei beschränkung bezüglich der grösse von dateien oder der länge von zeilen.

grep eine abkürzung von *„global/regular expression/print“* sucht in einer textdatei nach zeilen, die ein bestimmtes *pattern* beinhalten, das durch einen regulären ausdruck definiert ist.

awk benannt nach seinen autoren Aho, Weinberger und Kernighan, kann komplexe manipulationen auf textdateien vornehmen:

- Es zerlegt zeilen in „felder“, die – sofern nichts anderes spezifiziert wurde – durch *white space* voneinander getrennt sind,
- führt sodann „aktionen“ auf zeilen aus, in denen – ähnlich wie bei `grep` – ein bestimmtes *pattern* vorkommt. Dabei gilt:
  - Wenn kein *pattern* angegeben wurde, wird die aktion für jede zeile ausgeführt und
  - wenn keine aktion angegeben wurde, wird jede zeile, die auf das *pattern* passt, unmodifiziert ausgedruckt.
- Ausserdem besitzt `awk` „assoziative arrays“, das heisst also arrays, die auch durch zeichenketten indiziert werden können und nicht nur, wie sonst, durch zahlen.

Ein paar beispiele[36] sollen zeigen, was `awk` leisten kann. Grundlage ist eine kleine textdatei `coins.txt` mit angaben zu einer münzsammlung:

```
gold     1    1986  USA                 American Eagle
gold     1    1908  Austria-Hungary     Franz Josef 100 Korona
silver  10    1981  USA                 ingot
gold     1    1984  Switzerland         ingot
gold     1    1979  RSA                 Krugerrand
gold     0.5  1981  RSA                 Krugerrand
gold     0.1  1986  PRC                 Panda
silver   1    1986  USA                 Liberty dollar
gold     0.25 1986  USA                 Liberty 5-dollar piece
silver   0.5  1986  USA                 Liberty 50-cent piece
silver   1    1987  USA                 Constitution dollar
gold     0.25 1987  USA                 Constitution 5-dollar piece
gold     1    1988  Canada              Maple Leaf
```

Ein aufruf

```
awk '/gold/' coins.txt
```

leistet nichts anderes als ein `grep gold coins.txt`. Was `grep` aber nicht kann, ist folgendes:

```
awk '/gold/ {print $5,$6,$7,$8}' coins.txt
```

Hier druckt `awk` nur das fünfte bis achte feld der datei mit den textlichen beschreibungen der münzen aus, und zwar nur für die goldmünzen:

[36] von `http://www.vectorsite.net/tsawk_1.html`

```
American Eagle
Franz Josef 100 Korona
ingot
Krugerrand
Krugerrand
Panda
Liberty 5-dollar piece
Constitution 5-dollar piece
Maple Leaf
```

Ein beispiel für eine etwas komplexere aktion, die auf jede eingabezeile (kein pattern!) angewendet wird, wäre:

```
awk '{if ($3 < 1980) print $3,"   ",$5,$6,$7,$8,"(",$3,")"}' coins.txt
```

mit der Ausgabe

```
1908     Franz Josef 100 Korona
1979     Krugerrand
```

Wir haben aber im prinzip die volle sprache C zur verfügung für die aktionen:

```
awk '/gold/ {gold++;ounces += $2} END {print gold,"pieces of gold, total
weight",$ounces,"ounces"}' coins.txt
```

Dass die variablen `gold` und `ounces` hier weder deklariert noch initialisiert wurden, muss nicht stören, `awk` erledigt das für uns automatisch. Das pattern `END` ist vordefiniert: es wird wirksam, wenn die eingabedatei zu ende ist. (Natürlich gibt es dazu passend auch `BEGIN`.) Die ausgabe dieser anweisung ist dann

```
9 pieces of gold, total weight 6.1 ounces
```

Für komplexere anwendungsfälle ist es ratsam, das `awk`-programm in eine separate datei zu schreiben, etwa die folgende `coins.awk`:

```
/gold/    { num_gold++; wt_gold += $2 }       # Get weight of gold.
/silver/  { num_silver++; wt_silver += $2 }   # Get weight of silver.
END { val_gold = 485 * wt_gold;
# Compute value of gold.
      val_silver = 16 * wt_silver;
# Compute value of silver.
      total = val_gold + val_silver;
      print "Summary data for coin collection:";  # Print results.
      printf ("\n");
      printf ("   Gold pieces:                  %2d\n", num_gold);
```

```
printf ("   Weight of gold pieces:         %5.2f\n", wt_gold);
printf ("   Value of gold pieces:        %7.2f\n",val_gold);
printf ("\n");
printf ("   Silver pieces:                 %2d\n", num_silver);
printf ("   Weight of silver pieces:       %5.2f\n", wt_silver);
printf ("   Value of silver pieces:      %7.2f\n",val_silver);
printf ("\n");
printf ("   Total number of pieces:        %2d\n", NR);
printf ("   Value of collection:         %7.2f\n", total); }
```

Der aufruf

```
awk -f coins.awk coins.txt
```

liefert dann die ausgabe

```
Summary data for coin collection:

   Gold pieces:                    9
   Weight of gold pieces:          6.10
   Value of gold pieces:        2958.50

   Silver pieces:                  4
   Weight of silver pieces:       12.50
   Value of silver pieces:       200.00

   Total number of pieces:        13
   Value of collection:         3158.50
```

9.2 Das werkzeug „make“

Die im letzten kapitel geschilderte situation mit den schnittstellenbeschreibungen bei Modula-2 zeigt auf, dass es häufig komplexe abhängigkeiten zwischen dateien geben kann. Wird eine schnittstellenbeschreibung `M.def` geändert, so muss anschliessend die entsprechende symboldatei `M.sym` neu erzeugt werden. Wir sagen: `M.sym` hängt ab von `M.def`. Die kompletten spielregeln lassen sich wie folgt beschreiben:

- Eine symboldatei hängt ab von ihrem zugehörigen definitionsmodul sowie von den symboldateien aller in das definitionsmodul importierten module.
- Ein objektmodul hängt ab von seinem zugehörigen implementierungsmodul, seiner zugehörigen symboldatei sowie von den symboldateien aller in das implementierungsmodul importierten module.

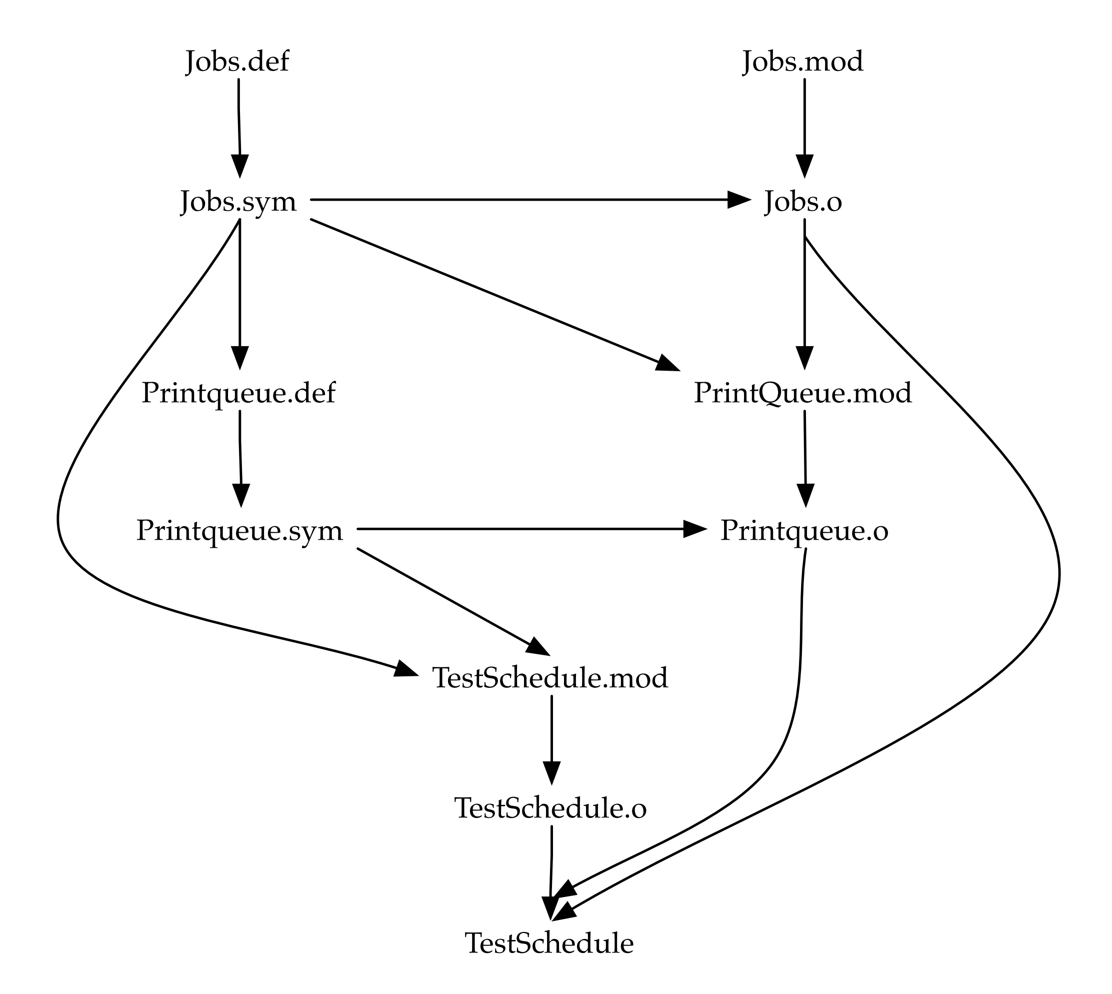


Abbildung 52: Abhängigkeiten im Printqueue-beispiel

```
# makefile for PrintQueue example
# requires GNU make

MODULES= Jobs PrintQueue

%.sym: %.def
	mc $<

%.o: %.mod %.sym
	mc -c $<

TestSchedule: TestSchedule.o $(MODULES:%=%.o)
	mc -o $@ $^

TestSchedule.o: TestSchedule.mod $(MODULES:%=%.sym)
	mc -c TestSchedule.mod

test: TestSchedule
	./$<

PrintQueue.sym: PrintQueue.def Jobs.sym

tar: PrintQueue.tgz

PrintQueue.tgz: $(MODULES:%=%.mod) $(MODULES:%=%.def) TestSchedule.mod
	tar -czvf $@ $^

clean:
	rm -f TestSchedule *.o *.tgz

veryclean: clean
	rm -f *.sym
```

Abbildung 53: `makefile` für die druckerwarteschlange

- Ein ausführbares programm hängt ab von den objektmodulen aller importierten module.

Abb. 52 zeigt die abhängigkeiten für das *PrintQueue*-beispiel aus abschn. 2.5. Zu beachten ist dabei, dass wir es im grunde nur mit drei modulen zu tun haben, auch wenn diese hier in form von 11 dateien auf uns zutreten. Die abhängigkeiten sind schon in diesem wirklich simplen beispiel durchaus komplex zu nennen.

Ein system aus vielen modulen lässt sich deshalb kaum ohne werkzeuge vernünftig pflegen. Unix hat ein entsprechendes werkzeug anzubieten, nämlich das programm `make` von Stuart Feldman (1979). Das programm `make` liest aus einem sogenannten `makefile` beschreibungen der abhängigkeiten zwischen dateien und führt daraufhin alle jeweils notwendigen aktionen durch. Einträge („*stanzas*“ = „strophen“) in diesem `makefile` haben im wesentlichen die folgende struktur: Zunächst wird ein gewisses *ziel* aufgelistet („*target*“), welches daran zu erkennen ist, dass dahinter ein doppelpunkt folgt. In der regel ist das ziel eine datei, die durch den `make`-prozess hergestellt werden soll; wir werden jedoch sehen, dass es auch sogenannte *pseudoziele* gibt, bei denen andere aktionen ausgeführt werden. Hinter dem doppelpunkt folgt dann eine liste von sogenannten *abhängigkeiten* („*dependants*“); dies sind die dateien, von denen das ziel abhängt. Unter der zeile mit den beschreibungen des ziels und der abhängigkeiten findet sich dann eine *aktion*, die ausgeführt werden soll, wenn eine der abhängigkeiten jünger ist als das ziel. Diese aktionszeile muss immer mit einem tabulator (HT, ASCII-Zeichen 9) beginnen; dies ist ein berüchtigter problempunkt, wenn makefiles mit e-mail verschickt werden, denn dabei werden meist tabulatoren durch leerstellen ersetzt.

Abb. 53 zeigt ein `makefile` für das *PrintQueue*-beispiel. Hier werden allerdings konstrukte des *GNU make* verwendet, die über das ursprüngliche `make` von Stuart Feldman hinausgehen.

In den ersten zeilen wird nur eine variable `MODULES` definiert, welche als abkürzung für die hier beteiligten module dient und es werden allgemeine regeln für symboldateien und objektmodule von implementierungsmodulen beschrieben[37]. Hundertprozentig hilfreich ist die regel für die objektmodule allerdings noch nicht, da sie nicht auf die importe in den definitionsmodulen eingeht. Die „automatische variable“ `$<` steht dabei für die erste (und in diesem fall einzige) abhängigkeit.

Die erste „echte“ strophe ist die für das ausführbare programm *TestSchedule*. Sie beschreibt, dass dieses von `TestSchedule.o` und den objektmodulen aller module aus der variablen `MODULES` abhängt. Die komplizierte wendung mit

[37]Für einige typen von dateien, z.B. C-programme, sind solche regeln nicht nötig, da `make` für diese bereits über eingebaute regeln verfügt.

den prozentzeichen innerhalb der klammer ist ein sog. *modifikator*, der hier angibt, dass die einzelnen namen innerhalb von MODULES am ende jeweils durch .o ergänzt werden sollen. Deshalb wirkt diese zeile gerade so, als ob wir hier

```
TestSchedule: TestSchedule.o Jobs.o PrintQueue.o
```

geschrieben hätten. Die in abb. 53 gewählte version ist aber — unter der voraussetzung, dass die variable MODULES richtig definiert wurde — zu bevorzugen, da es dabei nicht so leicht vorkommt, dass eine abhängigkeit vergessen wird. In der tat taucht die variable MODULES an mehreren stellen in diesem makefile auf. Wäre hier überall die expandierte form angegeben, so wäre es ziemlich wahrscheinlich, dass bei änderungen in der projektstruktur irgendwo vergessen würde, die weiter notwendigen module hinzuzufügen. Dies ist sowieso eine der grundlegenden regeln für das programmieren: Alles, was mehrfach auftaucht, soll unbedingt durch eine einzige definition „herausfaktorisiert“ werden.

In der aktionszeile bezeichnet $@ das ziel und $^ die folge aller abhängigkeiten, durch leerstellen getrennt.

Interessant ist sicher die strophe zu PrintQueue.sym: Hier ist gar keine aktionszeile angegeben. Diese ist auch nicht notwendig, da bereits die zu beginn aufgeführte regel die nötige aktion enthält. Die strophe hat hier nur den einzigen zweck, die zusätzliche abhängigkeit von Jobs.sym zu erfassen, die sich aus dem import von Jobs in das definitionsmodul von PrintQueue ergibt.

Ein aufruf von make ohne zusätzliche parameter führt dazu, dass das im makefile zuerst genannte ziel *(primary target)* entsprechend der darunter spezifizierten aktion einmalig herzustellen versucht wird. Man kann aber auch z. b. durch den Aufruf make TestSchedule.o ein anderes ziel anstatt des ersten erzeugen. In dem zuerst genannten beispiel würde make zur feststellung der frage, ob TestSchedule älter ist als eine seiner abhängigkeiten, zunächst einmal versuchen, diese abhängigkeiten aufzusuchen. Falls diese — was mindestens beim ersten programmlauf auftritt — selbst nicht existieren, würde make zunächst rekursiv versuchen, diese abhängigkeiten zu *erzeugen*. Make stellt in der tat zunächst eine topologisch sortierte liste aller dateiabhängigkeiten her, so dass die notwendigen aktionen in jedem fall in einer logischen reihenfolge und nur ein einziges mal durchgeführt werden können.

In unserem beispiel wäre beim ersten make nach der erstellung der quelldateien die folgende reihenfolge von aktionen sinnvoll:

```
mc Jobs.def
mc PrintQueue.def
mc -c TestSchedule.mod
```

```
mc -c Jobs.mod
mc -c PrintQueue.mod
mc -o TestSchedule TestSchedule.o Jobs.o PrintQueue.o
```

Eine besonderheit sind die *pseudoziele* (im obigen beispiel `test`,`tar`,`clean`, `veryclean`). Hier ist überhaupt nicht daran gedacht, irgendeine datei zu erzeugen, sondern es sollen nur die nachfolgenden aktionen ausgeführt werden. Im beispiel `clean` ist dies die entfernung aller objektdateien und des ausführbaren programms. Beachte, dass das „ziel", eine datei mit dem namen `clean` herzustellen, durch diese aktion nicht erfüllt wird! Eine erweiterte stufe des aufräumens beschreibt das ziel `veryclean`. Hier wird zunächst einmal `clean` „gemacht", d. h. die entfernung aller objektdateien; dann werden jedoch zusätzlich auch noch die symboldateien entfernt. Ein aufruf von `make test` sorgt dafür, dass alle für den programmtest nötigen dateien hergestellt bzw. aktualisiert werden und das testprogramm anschliessend im günstigen fall auch gleich gestartet wird.

Der arbeitszyklus des typischen Unix-programmierers lässt sich in der tat durch die schleife „think–edit–make–debug" zutreffend beschreiben.

`make tar` sorgt dafür, dass die jeweils aktuelle version der quelldateien in einem archiv namens `PrintQueue.tgz` abgelegt werden. Ein wiederholter aufruf von `make tar`, ohne dass zwischendurch eine der quelldateien berührt wurde, führt nur zu der meldung „`Target "tar" is up to date.`" ohne irgendwelche aktionen.

Kernpunkt der ganzen anwendung von `make` ist selbstverständlich, dass der `makefile` korrekt erstellt wurde. Werden nicht alle abhängigkeiten durch den `makefile` wirklich wiedergegeben, so wird auch der aufruf von `make` nicht in allen fällen das gewünschte resultat zeigen. Im fall einer softwaretechnik-programmiersprache wie z. B. Modula-2 lassen sich jedoch alle informationen über die dateiabhängigkeiten aus den modultexten selbst ablesen. Es gibt deshalb softwarewerkzeuge, welche aus einem programmodul ein geeignetes `makefile` konstruieren. Das geht bei Modula-2 recht gut. Bei sprachen wie C, wo man nicht ausdrücklich sagen muss, *was* man importiert, geschweige denn *von woher*, wird die sache schwieriger. IDEs kümmern sich gewöhnlich um alle arten von abhängigkeiten und machen dann automatisch alles richtig.

9.3 Das werkzeug „Ant"

Ein modernerer ansatz zum gleichen problem ist *Ant*, siehe z. B. Serrano und Ciordia (2004). Die beschreibung der abhängigkeiten ist hier in einer *build-datei* enthalten, die in XML geschrieben ist. (Normalerweise heisst sie `build.xml`.) Ant entstand im zusammenhang mit dem Apache-projekt, ist eindeutig auf

Java als programmiersprache zugeschnitten und macht sich von der vorstellung frei, dass hier unbedingt ein Unix-ähnliches betriebssystem zum einsatz kommen muss. Ausserdem nimmt Ant rücksicht auf die tatsache, dass Javaprojekte typischerweise mit einer ganzen reihe unterschiedlicher verzeichnisse *(directories)* zu tun haben, während `make` im grunde nur auf die anwendung innerhalb eines einzigen verzeichnisses eingerichtet ist. In einem typischen `makefile` wird man deshalb beobachten, dass `make` rekursiv in allerlei verzeichnisse einsteigt und die dort liegenden `makefiles` separat ausführt.

Abgesehen davon, dass eine Ant-build-datei dem projekt als ganzem einen namen gibt, können hier bestimmte eigenschaften *(properties)* global definiert werden; jedes einzelne *target* wird durch einen XML-teilbaum beschrieben, kann dabei wie das ganze projekt auch mit einer kurzen beschreibung versehen werden und eine beliebige menge von *tasks* beinhalten. Eine ganze menge von solchen *tasks* ist bereits vordefiniert (darunter die zu erwartenden vorrichtungen zum kopieren und löschen von dateien, zum compilieren, anlegen von verzeichnissen etc.), weitere tasks kann der programmierer selbst in form von Java-klassen hinzufügen.

Der preis für die erhöhte flexibilität von Ant ist naturgemäss eine grössere komplexität des werkzeugs, das einen ungleich höheren einarbeitungsaufwand erfordert als `make`.

9.4 Konfiguration von software

Leider ist die situation in wirklichkeit noch viel komplizierter, als dass sie mit `make` allein gelöst werden könnte, da praktisch jede software eingebettet ist in ein komplexes system aus anderen softwareprodukten. Unter einer *plattform* verstehen wir eine kombination von maschine und betriebssystem; der allgemeinere begriff *substrat* bezieht auch noch weitere gegebenheiten wie etwa eine graphische oberfläche, ein fenstersystem und anderes ein. Je nach substrat gibt es womöglich andere optionen für compiler und linker, andere pfade zu programmbibliotheken, ja sogar andere namen für compiler, programmbibliotheken etc. Solche gegebenheiten beeinflussen natürlich das `makefile`; es ist deshalb durchaus nicht ungewöhnlich, den prozess der erzeugung eines lauffähigen programms mit einem

```
make makefile
```

zu beginnen, um auch abhängigkeiten ausserhalb des gerade betrachteten softwarepakets erfassen zu können.

Leider haben die meisten compiler, bibliotheken und betriebssysteme auch bekannte fehler, die dazu führen, dass bestimmte teile von programmen je nach plattform unterschiedlich abgefasst werden müssen. Die meisten pro-

```
Files used in preparing a software package for distribution:

     your source files --> [autoscan*] --> [configure.scan] --> configure.ac

     configure.ac --.
                    |   .------> autoconf* -----> configure
     [aclocal.m4] --+---+
                    |   `-----> [autoheader*] --> [config.h.in]
     [acsite.m4] ---'

     Makefile.in -------------------------------> Makefile.in

Files used in configuring a software package:

                            .-------------> [config.cache]
     configure* ------------+-------------> config.log
                            |
     [config.h.in] -.       v            .-> [config.h] -.
                    +--> config.status* -+               +--> make*
     Makefile.in ---'                    `-> Makefile ---'
```

Abbildung 54: GNU distribution

grammiersprachen verfügen über mechanismen zur *bedingten compilation*; in der Sprache C sind diese durch den präprozessor und seine handhabung von *variablen* durch `#define` und `#ifdef` dargestellt. Der gleiche mechanismus kann auch dazu verwendet werden, verschieden leistungsfähige versionen des gleichen programms (z. b. evaluationsversion, vollversion) herzustellen. Damit dieser mechanismus funktioniert, müssen natürlich vor dem compilationsprozess alle diese präprozessorvariablen richtig gesetzt werden.

Ein weiteres problem, das üblicherweise zu lösen ist, ist der umgang mit *installationsprozeduren und -hilfen*, die ebenfalls von plattform zu plattform stark unterschiedlich sind.

Software, die für viele plattformen im source code verteilt wird, verfügt deshalb in der regel über recht elaborierte `configure`-skripte. Komplexe beispiele hierzu finden sich in praktisch jedem paket der GNU software.

Abb. 54 zeigt den workflow bei der distribution von GNU software.

Jede(r) informatiker(in) sollte sich – am besten gleich im zusammenhang mit dieser vorlesung! – diesen elaborierten mechanismus einmal detaillierter angeschaut haben; man braucht es im richtigen leben schneller, als man denkt, und es lässt sich unvernünftig viel zeit damit zubringen, wenn man in den falschen dateien herumeditiert!

Als informatiker sollten wir uns nicht wundern, dass jedes problem gleich

auf einer höheren hierarchieebene erneut auftaucht: So werden also nicht nur, wie bereits angedeutet, die makefiles häufig automatisch mit hilfe von make selbst erzeugt, sondern schon die erste vorlage eines makefile wird von einem configure-skript aus einem „rahmen“ makefile.in erzeugt, der seinerseits vermutlich durch ein programm automake aus makefile.am erzeugt wurde. Aber auch das configure-skript seinerseits wird aus einem mitgelieferten „rahmen“ configure.in, welcher die eigenheiten des gegenwärtig betrachteten systems beschreibt, automatisch erzeugt. Hierzu gibt es ein eigenes programm (autoconf). Es hat deshalb keinen sinn, im makefile dinge zu ändern, weil sie beim nächsten aufruf von configure verloren gehen. Ebenso hat es keinen sinn, in makefile.in dinge zu ändern, die in makefile.am auch vorkommen, denn diese werden beim nächsten aufruf von automake verschwinden. Es ist auch völlig sinnlos, in dem configure-skript etwas ändern zu wollen (ganz abgesehen davon, dass dafür der schwarze gürtel in der shell-programmierung nötig wäre), denn der nächste aufruf von autoconf wird diese änderungen vernichten.

Die configure-skripte leisten im wesentlichen das folgende:

- Prüfen der vorliegenden plattform, d. h. hardware, betriebssystem, compiler etc.
- Auffinden der suchpfade für programmquellen und bibliotheken
- Auffinden weiterer wichtiger details des substrats (compiler, linker, fenstersystem, installationsprozeduren, hilfsprogramme, . . .)
- Setzen aller relevanten präprozessorvariablen
- Erzeugung von makefiles aus vorgegebenen rahmen

9.5 Versionshaltung mit SVN

Jedes programm, das wirklich eingesetzt wird, wird auch ständig geändert. Ein geordneter softwaretechnikprozess setzt voraus, dass solche änderungen protokolliert werden: Was wurde geändert, wann, warum und von wem. Im prinzip könnte man dies natürlich in form von kommentaren in den quellen vermerken, allerdings würden dann die quellen sehr bald ziemlich unübersichtlich, das eigentliche programm wäre womöglich nur noch schwer zwischen den kommentaren wiederzufinden. Möchte man überdies zur fehlersuche alte versionen der programme rekonstruieren (z. b. diejenigen, die bei einem bestimmten kunden laufen), so helfen solche kommentare wenig, da sie nicht zweifelsfrei die rekonstruktion ermöglichen.

Eine möglichkeit, die ebenfalls unzufriedenstellend ist, besteht in der abspeicherung kompletter systemabzüge in archiven (wie im letzten abschnitt im `makefile` angedeutet). Dies stellt einerseits eine ernorme platzvergeudung dar, da ausser den geänderten teilen auch alle unveränderten im archiv enthalten sind (und enthalten sein müssen!), andererseits beantwortet es nicht die frage nach den urhebern von änderungen und es schafft keine übersicht. Der kernel von SunOS 4.0 besteht beispielsweise aus über 1000 dateien in mehreren dutzenden von verzeichnissen; eine versionshaltung für solch ein system lässt sich nicht durch systemabzüge verwirklichen.

Zu bedenken ist ausserdem, dass softwareentwicklung heute häufig verteilt im netz stattfindet und nicht mehr wie früher auf einem einzigen rechner. Die zusammenarbeit in open source projekten ist in der tat einer der gründe, warum versionshaltungssysteme in den letzten jahren wieder stark an bedeutung zugenommen haben.

Der kerngedanke eines versionshaltungssystems ist es, eine zentrale stelle zu schaffen, an der *alle* versionen einer software abgelegt sind, das sogenannte repositorium (*repository*). Das repositorium ist nach dem paradigma einer *leihbibliothek* organisiert: Hier kann man komponenten ausbuchen (*checkout*) und (geändert) wieder einbuchen (*checkin*) und ausserdem informationen über den änderungsstand und andere verwaltungsdetails ablegen und erhalten. Ein bestimmter zustand einer datei heisst hier eine *revision*. Das versionshaltungssystem führt buch über revisionsnummern, änderungszeitpunkte und die benutzerkennzeichen der personen, die neue revisionen im repositorium ablegen. Für jeden speicherprozess wird ein *change comment* verlangt, der über art und grund der änderung auskunft gibt. Frühere versionshaltungssysteme sind in abschnitt C.3 beschrieben. Heute im praktischen einsatz sind hauptsächlich *Subversion (SVN)*, *Mercurial (HG)*[38] und *Git*; diese folgen zwei unterschiedlichen strategien:

Zentrales repositorium SVN verwendet ein einziges zentrales repositorium, von dem alle entwickler sich arbeitskopien ziehen. Wenn ein entwickler einen seiner meinung nach stabilen zustand erreicht hat, speichert er die änderungen zurück ins repositorium („*commit*“). Damit ist die änderung dann für alle sichtbar, die sich mit einem befehl *update* den letzten zustand abholen können. Typisch für SVN-verwaltete projekte ist es, dass das repositorium verschiedene zweige (alternative versionen, *branches*) eines systems enthält, die separat weiterentwickelt werden und später wieder in den „Stamm“ (*trunk*) zusammengeführt („gemischt“) werden. Die *branches* sind für alle sichtbar.

Verteilte repositorien Bei Mercurial und Git dagegen hat jeder entwickler ein

[38] Hg ist das in der chemie übliche symbol für quecksilber („mercurial“)

komplettes repositorium auf seinem rechner; dieses erhält er, indem er sich von einer anerkannten quelle einen „ableger“ (*clone*) besorgt. In seinem eigenen repositorium kann er dann unabhängig von allen anderen arbeiten, kann revisionen abspeichern (*„commit“*) und wieder rückgängig machen, ohne dass dies die anderen entwickler bemerken. Die notwendigkeit für *branches* entfällt damit. Wenn er später denkt, auch die anderen entwickler sollten von seinen änderungen profitieren können, kann er seinen zustand auf den rechner hochladen (*„push“*), von wo er den ableger gezogen hat, vorzugsweise, nachdem er zunächst (mit *„pull“*) von dort den letzten allgemein verfügbaren zustand abgeholt und mit *„merge“* mit seinem lokalen zustand abgeglichen hat.

SVN vergibt für jedes *commit* eine neue revisionsnummer, die dann für das ganze projekt gilt. O'Sullivan (2009) beschreibt die tücken, die dabei auftreten können: Angenommen, Alice und Bob besorgen sich beide die revision 105 aus dem repositorium und entwickeln diese weiter. Wenn Alice dann ihre änderungen speichert, wird dies revision 106, und wenn Bob danach abspeichern will, wird ihm dies verweigert, weil die im repositorium befindliche revision 106 verschieden ist von der revision 105, auf der seine änderungen basieren. Bob muss also zuerst seine dateien auf den zustand 106 bringen, bevor er seine revision (107) abspeichern kann. Sollten seine änderungen im konflikt mit Alices änderungen stehen, muss er erst die konflikte bereinigen, aber das problem entsteht, wenn Alice an ganz anderen stellen gearbeitet hat als Bob: dann geht das mischen klaglos vonstatten, aber jetzt befindet sich im repositorium ein softwarezustand, den in dieser form noch niemand gesehen geschweige denn getestet hat!

Das zentrale konzept bei SVN sind also *revisionen*, Mercurial dagegen konzentriert sich auch *„changesets“*, d. h. auf die menge der änderungen zwischen einer revision und einer anderen. Das wirkt zunächst wie ein streit um des kaisers bart, hat aber wichtige auswirkungen, wie Joel Spolski (2010) in seinem tutorial bemerkt: Während SVN nur weiss, dass es sich bei einem *commit* um zwei verschiedene revisionen eines projekts handelt und dann versucht, diese irgendwie zu mischen, hat Mercurial bereits einen überblick darüber, was sich konkret alles geändert hat und kann von daher besser einen konsistenten zustand herstellen. Die anzahl der *merge conflicts* wird also drastisch reduziert.

9.6 Werkzeuge zur dokumentation

In diesem zusammenhang muss es selbstverständlich auch um die werkzeugunterstützung für die dokumention gehen. Dabei muss zuerst einmal geklärt werden, was „dokumentation“ eigentlich bedeuten soll. Weitgehend ausklammern wollen wir hier den bereich der dokumentation für den benutzer (hand-

bücher etc.), da dies ein spezialbereich ist, der neben technischen kenntnissen auch „schriftstellerische“ fähigkeiten erfordert. In professionell geführten unternehmen wird die benutzerdokumentation von eigenen abteilungen hergestellt, oft parallel zum prozess der softwareentwicklung. Das mag vielleicht ein grund dafür sein, dass die benutzerhandbücher manchmal nicht hundertprozentig zur software passen.

In dieser vorlesung kann es uns nur um die *technische dokumentation* gehen, die hauptsächlich unter dem gesichtspunkt der weiterentwicklung und wartung von programmsystemen geschrieben wird. Die frage, *wann* der richtige zeitpunkt ist, eine solche technische dokumentation zu schreiben, ist schnell beantwortet: In einem geordneten software-entwicklungsprozess wird in jedem fall *zuerst* dokumentation geschrieben, bevor überhaupt mit der programmierung begonnen wird. Dazu können graphische darstellungsmittel wie die in kapitel 4 vorgestellten dienen; vor allem aber gehören eine ganze menge von *texten* dazu. Während des programmierprozesses muss diese dokumentation stetig überprüft, möglicherweise geändert und vor allem erweitert werden, z. b. um details von datenstrukturen und algorithmen. Jede entwurfsentscheidung gehört sorgfältig dokumentiert, inklusive der angabe des verantwortlichen autors mit datum und uhrzeit. Auf diese weise wird — wenn genügend disziplin entfaltet wird und mit jeder änderung/erweiterung von programmen gleich die entsprechende änderung/erweiterung der dokumentation erfolgt — gewährleistet, dass die dokumentation wirklich zum programm passt und dass ihre erstellung nicht als unangenehme, langweilige „aufräumaufgabe“ nach abschluss des programmierprozesses verstanden werden kann.

Im folgenden werden einige werkzeuge vorgestellt, die zur erstellung technischer dokumentationen dienen können.

9.6.1 Nroff

Technische dokumentationen sind zuerst und vor allem texte. Um einen text lesbar zu gestalten, sind textverarbeitungsprogramme unersetzlich. Getreu der Unix-ideologie gibt es hierzu ein werkzeug, das je nach lokaler kultur `nroff`, `troff`, `groff` oder ähnlich heisst[39]. Im grunde handelt es sich dabei um nicht viel mehr als einen allgemeinen makroprozessor zusammen mit diversen sammlungen vordefinierter makros. Mit dem aufruf

```
nroff -man
```

bindet man etwa ein makropaket ein, das auf die produktion von Unix manual pages („online-handbücher“) spezialisiert ist. Diese manual pages sind

[39]Nroff war eine abkürzung für „new runoff“, troff ist eine variante davon, die mit phototypesettern arbeiten kann.

ein zwischending zwischen technischer dokumentation und der hier ausgeklammerten benutzerdokumentation: Sie verraten nicht wirklich viel über die arbeitsweise der dokumentierten programme, aber sie geben in kondensierter und von der form her weitgehend normierter fassung auskunft über die einsatzmöglichkeiten, einschränkungen und parameter (ja, sogar: bekannte fehler) von Unix-programmen auskunft.

Getreu der Unix-ideologie sind die nroff-quelltexte ganz normale textdateien, die mit kommandos durchsetzt sind, welche der nroff-prozessor interpretiert. Dieses vorgehen hat den vorteil, dass sich auch die texterzeugung in eine werkzeugkette einbetten lässt, welche spezielle vorrichtungen anbietet, die in einem bestimmten einsatzkontext benötigt werden, von einer allgemeineren sichtweise her aber entbehrlich sind. So gibt es also z. b. eigene programme, `eqn` zum satz von gleichungen, `tbl` zum satz von tabellen, `pic` zum erzeugen von abbildungen, die sich mit nroff in eine werkzeugkette kombinieren lassen. Jedes dieser werkzeuge liest die an es selbst gerichteten kommandos in einer eingabedatei und reagiert darauf dadurch, dass es den entsprechenden teil der datei verändert; alles andere wird unverändert weitergegeben.

Die nroff-kommandos erkennt man daran, dass sie am anfang einer zeile stehen und mit einem punkt beginnen, beispielsweise:

.SH Definition einer zwischenüberschrift (*section heading*)
.PP Anfang eines neuen abschnitts (*paragraph*)
.NH n Numerierte überschrift (*numbered heading*) der schachtelungstiefe n
.br Zeilenende (*break*)
.bp Seitenende (*begin page*)

Abb. 55 zeigt einen ausschnitt aus einer UNIX man page und die entsprechende `nroff`-Quelle dazu.

Ein werkzeug mit einer ganz ähnlichen ideologie ist TeX bzw. LaTeX; der unterschied ist — abgesehen von einer radikal verschiedenen syntax — dass TeX den anspruch erhebt und erfüllt, wirklich professionellen buchsatz (inklusive eines ästhetisch befriedigenden satzes mathematischer formeln) herzustellen, während `nroff` mehr auf den einsatz in einer werkzeugkette vorgerichtet ist.

9.6.2 Texinfo

Zum editor Emacs (in seinen verschiedenen versionen) gehört ein dateiformat mit zugehörigem werkzeug namens `texinfo`. Auch dies ist für die benutzerdokumentation gedacht, und zwar mit einer doppelten zielrichtung: einerseits gibt es die möglichkeit, bildschirmtaugliche hypertexte mit suchfunktion, navigation durch menüs und blättern mit hilfe von `Up`, `Next`, `Previous` zu er-

```
u-084-c008.eap.uni-tuebingen.de.klaeren:~ — u-084-c008.eap.uni-tu...

LOCALE(1)                 BSD General Commands Manual                LOCALE(1)

NAME
     locale -- display locale settings

SYNOPSIS
     locale [-a|m]
     locale [-ck] name [...]

DESCRIPTION
     locale displays information about the current locale, or a list of all
     available locales.

     When locale is run with no arguments, it will display the current source
     of each locale category.

     When locale is given the name of a category, it acts as if it had been
     given each keyword in that category.  For each keyword it is given, the
     current value is displayed.

OPTIONS
     -a        Lists all public locales.
:
```

```
.Dd August 27, 2004
.Dt LOCALE 1
.Os Darwin
.Sh NAME
.Nm locale
.Nd display locale settings
.Sh SYNOPSIS
.Nm
.Op Fl a|m
.Nm
.Op Fl ck
.Ar name
.Op ...
.Sh DESCRIPTION
.Nm
displays information about the current locale, or a list of all available
locales.
.Pp
When
.Nm
is run with no arguments,
it will display the current source of each locale category.
.Pp
When
.Nm
is given the name of a category,
it acts as if it had been given each keyword in that category.
For each keyword it is given, the current value
is displayed.
.Sh OPTIONS
.Bl -tag -width -indent
.It Fl a
Lists all public locales.
```

Abbildung 55: Unix man page

```
@dircategory Editors
@direntry
* VIPER: (viper).       The newest Emacs VI-emulation mode.
                          (also, A VI Plan for Emacs Rescue
                           or the VI PERil.)
@end direntry

@finalout

@titlepage
@title Viper Is a Package for Emacs Rebels
@subtitle a Vi emulator for Emacs
@subtitle November 2008, Viper Version 3.11.2

@author Michael Kifer (Viper)
@author Aamod Sane (VIP 4.4)
@author Masahiko Sato (VIP 3.5)

@page
@vskip 0pt plus 1filll
@insertcopying
@end titlepage

@contents

@ifnottex
@node Top, Overview,, (DIR)

@unnumbered Viper

We believe that one or more of the following statements are adequate
descriptions of Viper:

@example
Viper Is a Package for Emacs Rebels;
it is a VI Plan for Emacs Rescue
and/or a venomous VI PERil.
@end example
```

Abbildung 56: Texinfo

zeugen, andererseits kann man aus den gleichen quellen über TEX als umweg ein gedrucktes handbuch erstellen. Abb. 56 zeigt ein beispiel aus einer datei `viper.texi`; wie die aufbereitete seite in der online-dokumentation aussieht, kann man sehen, indem man im Emacs `M-x info RET` tippt, dann `m viper RET`.

9.6.3 Javadoc

Ganz deutlich auf die erzeugung technischer dokumentation ausgerichtet ist *Javadoc*, welches wiederum gleichzeitig der name eines werkzeugs und eines dateiformats ist. *Javadoc* ist dem sogenannten *single source*-prinzip verpflichtet, d. h. die dokumentation steht in der gleichen datei wie das zu dokumentierende programm. Ein solches vorgehen hat den vorteil, dass die konsistenz zwischen dokumentation und programm leichter zu erreichen ist, da letztlich nur jeweils eine datei angeschaut und modifiziert werden muss. Die voraussetzung dafür ist, dass die dokumentation für den compiler nicht sichtbar ist; dies ist leicht zu erreichen, indem man sie in kommentaren unterbringt. Die normale kommentarbegrenzung in Java ist `/* */`, compiler überlesen alles, was sich dazwischen befindet. Die Javadoc-kommentare haben einen stern mehr: `/** */`, den der compiler schon nicht mehr sieht. Das werkzeug Javadoc erzeugt aus Java-programmen HTML-dateien mit dokumentation, die mit jedem browser angeschaut werden können. Daraus ergibt sich, dass innerhalb der Javadoc-dokumentation jede HTML-konstruktion verwendet werden kann, inklusive hyperlinks auf andere seiten.

Javadoc selbst kennt eine reihe von markierungen (*tags*), die gesondert behandelt werden und die zum teil bindend verlangt werden; die wichtigsten davon sind:

@author Autor (nur bei klassen und schnittstellen, bindend erforderlich)

@version Version (nur bei klassen und schnittstellen, bindend erforderlich)

@param Parameter von methoden und konstruktoren

@return Rückgabewert von methoden

@throws Ausnahmesituationen

@see Querverweis

Es gibt bestimmte richtlinien, wie Javadoc-kommentare vernünftigerweise geschrieben werden sollen; z. b. ist die obige reihenfolge von markierungen verpflichtend; siehe hierzu `http://java.sun.com/j2se/javadoc/writingdoccomments/index.html`.

9.6.4 Doxygen

TODO: Hier etwas über Doxygen sagen, www.stacks.nl/ dimitri/doxygen/, siehe auch Neville-Neil (2008a).

Kontrollaufgaben

1. Erläutern sie die funktionsweise des programms `make`!
2. Welche eigenschaften muß eine programmiersprache haben, damit sich `makefiles` automatisch aus den programmquellen generieren lassen?
3. Was versteht man unter der konfiguration von software?
4. Was leisten `configure`-skripte?
5. Was leisten versionshaltungssysteme? Wie funktionieren sie im prinzip?
6. Was ist der grundlegende unterschied zwischen SCCS und seinen nachfolgern?
7. Wodurch unterscheidet sich CVS von seinen vorgängern?

10 Free/Libre/Open Source Software

In diesem kapitel soll es hauptsächlich um software-lizenzmodelle für „offene software" gehen, wobei dieser begriff erstaunlich schwer zu fassen ist. Es kann dazu hilfreich sein, einen kurzen blick in die computergeschichte zu werfen. Details dazu finden sich bei Klaeren (2006).

Sieht man von den frühen experimenten von Konrad Zuse ab, so wurden die ersten rechner während des 2. weltkrieges für militärische zwecke gebaut und eingesetzt. Auch einer der ersten technisch-wissenschaftlichen rechner nach dem krieg, die am 29. April 1952 vorgestellte IBM 701, war allgemein als der „defense calculator" bekannt, weil sie im verlauf des koreakriegs aufgrund eines persönlichen gesprächs von T. J. Watson mit dem amerikanischen präsidenten Harry Truman für das verteidigungsministerium hergestellt worden war.

Zur damaligen zeit wurden computer als eine art von rechenmaschinen betrachtet und logischerweise ohne jegliche software ausgeliefert. Die verwender der maschinen – in der regel mathematiker – standen daher allesamt vor ganz ähnlichen problemen. Das wichtigste problem war es, dass das programmieren in der maschinensprache durch das eintippen von oktalzahlen so beschwerlich ist, dass doch zumindest ein symbolischer *assembler* benötigt wird, um halbwegs vernünftig arbeiten zu können. So ein assembler bietet *mnemonics* zur bezeichnung von instruktionen und *symbolische adressen* für das programmieren von sprüngen an.

Schon 1955 wurde deshalb eine benutzerorganisation SHARE (*Society for Help to Avoid Redundant Efforts*) gegründet, deren mitglieder – anfänglich 17 institutionen mit einer IBM 704, dem nachfolgemodell der 701 – sich gegenseitig die von ihnen geschriebene notwendige software zur verfügung stellten. Dies geschah, wie es allgemein der tradition wissenschaftlicher publikationen entspricht, unter wahrung des urheberrechts und unter ausschluss sämtlicher haftung. IBM unterstützte SHARE nicht nur ideell, sondern auch in bescheidenem umfang materiell und sah die älteste computerbenutzerorganisation durchaus auch als ein instrument zur verkaufsförderung. Programme aus der SHARE-bibliothek erhielt man nicht nur als lauffähige binärprogramme, sondern auch im quelltext, damit sich jedes mitglied einerseits über die arbeitsweise der programme informieren konnte und, wenn gewünscht, andererseits auch an der weiterentwicklung mitarbeiten konnte. Die anfangszeit der computersoftware ist also durch das *open source*-prinzip charakterisiert.

Das ist auch nicht verwunderlich, denn die frühen computer waren sämtlich in den händen von *wissenschaftlern*, die sich in ihrer computerverwendung von dem paradigma der wissenschaften leiten liessen: wissenschaftler *publizieren* ihre erkenntnisse, die dann von allen anderen wissenschaftlern frei verwendet,

verbessert, weiterentwickelt und erneut der menschheit zur verfügung gestellt werden. Es gehört zum guten ton in der wissenschaft, jeweils die *quellen* anzugeben, auf die man sich gestützt hat. Jede fremde erkenntnis, jeden fremden text muss ich als wissenschaftler ordentlich zitieren. Es gibt aber darüber hinaus keine einschränkungen darüber, was ich mit den erkenntnissen anderer anstelle. Konkret gesprochen: wenn jemand aufbauend auf den wissenschaftlichen erkenntnissen des einen etwas ableitet, was diesem nicht gefällt oder gar seinen absichten zuwiderläuft, so gibt es für den ersten keine möglichkeit, dagegen vorzugehen. Der einzige schutz, den ein wissenschaftler vor seiner ansicht nach missbräuchlicher verwendung seiner erkenntnisse haben kann, ist, diese erkenntnisse für sich zu behalten. Das entspricht aber nicht dem in der wissenschaft üblichen vorgehen.

Schon im jahr 1957 hatte SHARE 47 mitglieder und es gab rund 300 geprüfte programme in einer von IBM verwalteten bibliothek, die von mitgliedern kostenlos bezogen werden konnte, darunter das SOAP (*Symbolic Optimal Assembly Program*, der oben erwähnte assembler) und das SOS (*Share Operating System*) für die IBM 709, die inzwischen die 704 abgelöst hatte. Gegen ende der 1960er jahre hatte IBM deutlich mehr als 50 prozent des computermarkts besetzt und die konkurrenten strengten am 17.1.1969 eine *antitrust*-klage an. Ein resultat dieses prozesses, der bis 1983 lief, war dann das sogenannte *unbundling*: IBM durfte nun die software zu seinen grossrechnern nicht mehr verschenken, sondern musste sie separat berechnen. Dazu waren nun erstmals auch *lizenzverträge* notwendig, welche (unter anderem) die weitergabe gekaufter programme verboten. Man kann ohne übertreibung sagen, dass der oberste amerikanische gerichtshof hiermit die softwareindustrie gegründet hat.

Sobald juristen das feld betreten, wird das leben um einige grössenordnungen komplizierter. Lizenzen (wörtlich: erlaubnisse) regeln im fall von software, was der empfänger damit tun darf und was ihm verboten ist. Grundlage hierfür ist das *urheberrecht* (im angelsächsischen sprachgebrauch: *copyright*), das von land zu land unterschiedlich ausgeformt ist. Das (deutsche) urheberrecht ist eine scharfe waffe, die beispielsweise auch dazu eingesetzt werden kann, dass der enkel eines architekten versucht, den abriss eines bahnhofsseitenflügels zu verhindern. In einem der zahlreichen Stuttgart-21-prozesse mussten die richter deshalb eine abwägung der rechtsgüter vornehmen und haben in diesem fall entschieden, dass das recht der allgemeinheit auf einen anderen bahnhof schwerer wog als das recht des längst verstorbenen urhebers auf die unversehrtheit seines werkes.

§1 des urheberrechtsgesetzes sagt:

> *Die Urheber von Werken der Literatur, Wissenschaft und Kunst genießen für ihre Werke Schutz nach Maßgabe dieses Gesetzes.*

und §2 führt dazu weiter aus:

> *(1) Zu den geschützten Werken der Literatur, Wissenschaft und Kunst gehören insbesondere:*
> *1. Sprachwerke, wie Schriftwerke, Reden und Computerprogramme;*
> *2. . . .*

Das urheberrecht erlischt erst 70 jahre nach dem tod des urhebers und lässt sich nach §29 weder verkaufen noch verschenken; es ist *„nicht übertragbar, es sei denn, es wird in Erfüllung einer Verfügung von Todes wegen oder an Miterben im Wege der Erbauseinandersetzung übertragen."*

Worin genau das (deutsche) recht des urhebers besteht, sagt §15:

> *(1) Der Urheber hat das ausschließliche Recht, sein Werk in körperlicher Form zu verwerten; das Recht umfaßt insbesondere*
> *1. das Vervielfältigungsrecht (§ 16),*
> *2. das Verbreitungsrecht (§ 17),*
> *3. das Ausstellungsrecht (§ 18).*
> *(2) Der Urheber hat ferner das ausschließliche Recht, sein Werk in unkörperlicher Form öffentlich wiederzugeben (Recht der öffentlichen Wiedergabe). Das Recht der öffentlichen Wiedergabe umfasst insbesondere*
> *1. das Vortrags-, Aufführungs- und Vorführungsrecht (§ 19),*
> *2. das Recht der öffentlichen Zugänglichmachung (§ 19a),*
> *3. das Senderecht (§ 20),*
> *4. das Recht der Wiedergabe durch Bild- oder Tonträger (§ 21),*
> *5. das Recht der Wiedergabe von Funksendungen und von öffentlicher Zugänglichmachung (§ 22).*
> *(3) Die Wiedergabe ist öffentlich, wenn sie für eine Mehrzahl von Mitgliedern der Öffentlichkeit bestimmt ist. Zur Öffentlichkeit gehört jeder, der nicht mit demjenigen, der das Werk verwertet, oder mit den anderen Personen, denen das Werk in unkörperlicher Form wahrnehmbar oder zugänglich gemacht wird, durch persönliche Beziehungen verbunden ist.*

Damit stehen dem urheber sehr weitgehende rechte in bezug auf sein werk zu; ein programmierer beispielsweise könnte unter berufung auf sein urheberrecht genau bestimmen, wer sein programm wann zu welchen zwecken und wie lange benutzen darf; er könnte auch sein programm „zurückverlangen" und dabei darauf bestehen, dass auch alle sicherungskopien gelöscht werden.

Eine völlig andere, wenn auch benachbarte thematik sind die *patente*; §1 des deutschen patentgesetzes sagt:

> *(1) Patente werden für Erfindungen auf allen Gebieten der Technik erteilt, sofern sie neu sind, auf einer erfinderischen Tätigkeit beruhen und gewerblich anwendbar sind.*

und absatz 3 führt dazu weiter aus

> *(3) Als Erfindungen im Sinne des Absatzes 1 werden insbesondere nicht angesehen:*
> *1. Entdeckungen sowie wissenschaftliche Theorien und mathematische Methoden;*
> *2. ästhetische Formschöpfungen;*
> *3. Pläne, Regeln und Verfahren für gedankliche Tätigkeiten, für Spiele oder für geschäftliche Tätigkeiten sowie Programme für Datenverarbeitungsanlagen;*
> *4. die Wiedergabe von Informationen.*

Wer ein patent erworben hat, kann jedem anderen untersagen, die entsprechende erfindung zu nutzen; diese schutzfrist läuft nach dem deutschen recht 20 jahre.

Der beginn des verkaufs lizensierter software überlappte sich zeitlich mit dem start der microcomputer-ära („heimcomputer“, „personal computer“). Während firmen, die grossrechner einsetzten, im wesentlichen keine andere wahl hatten, als ordnungsgemässe softwarelizenzen zu kaufen, waren die besitzer kleiner, privater computer eher geneigt, die programme kostenlos untereinander zu tauschen. Bill Gates ist vermutlich einer der ersten, die das kopieren von software als diebstahl gebrandmarkt haben; in einem offenen brief an die besitzer von homecomputern schrieb er schon 1976:

> „Wie die mehrheit der computeramateure wissen müsste, stehlen die meisten von euch ihre software. Für hardware muß man bezahlen, aber software ist etwas, das man teilen kann. Wen kümmert’s schon, ob die leute, die daran gearbeitet haben, bezahlt werden.“

In der tat fehlte und fehlt immer noch vielen privaten computerbenutzern das unrechtsbewusstsein beim kopieren von software. Schliesslich erleidet der hersteller der software im gegensatz zu einem ladenbesitzer, der durch ladendiebstahl geschädigt wird, keinen *direkten* schaden durch das kopieren: der schaden ist nur *indirekt* als entgangener gewinn zu erkennen.

Anderen scheinen die lizenzbedingungen bei der kommerziellen software eine zu starke einschränkung der rechte des erwerbers zu sein. In der (durch aufreissen der verpackung automatisch zu akzeptierenden) lizenz zu Microsoft Frontpage 2002 hiess es beispielsweise:

> „Sie dürfen diese Software nicht in Verbindung mit irgendeiner Stelle verwenden, die Microsoft, MSN, MSNBC, Expedia oder eins ihrer Produkte oder Dienste herabsetzt oder das geistige Eigentum oder andere Rechte dieser Parteien verletzt oder irgendein Staats-, Bundes- oder internationales Gesetz verletzt oder Rassismus, Haß oder Pornographie verbreitet."

Versteckt zwischen sowieso selbstverständlichen gesetzlichen verboten – die sicher nur mit dem zweck aufgeführt wurden, die eigentliche absicht des paragraphen zu verschleiern – findet sich hier ein passus, der letztlich verhindert, dass ein kunde kritik an der lieferfirma äußert. Kein hersteller einer schreibmaschine hat jemals versucht, seine kunden dazu zu verpflichten, auf dieser maschine nichts negatives über den hersteller zu tippen.

Einschränkungen der benutzerrechte von lizensierter software veranlassten auch Richard Stallman im jahr 1984, seinen sicheren arbeitsplatz am MIT aufzugeben, um ein jahr später die *Free Software Foundation* zu gründen. Die wahl dieses namens war allerdings höchst problematisch, weil das englische wort „free"nicht nur „frei", sondern auch „kostenlos" bedeutet. Stallman bemüht sich deshalb, stets zu betonen: „‚free' as in ‚free speech', not as in ‚free beer'". Das französische wort *libre* hat nicht die gleiche doppelbedeutung wie das englische *free*; deshalb verwenden viele der deutlichkeit halber statt *free software* den begriff FLOSS *(Free/Libre Open Source Software)*.

Stallman gibt im „GNU Manifesto" [40] die folgende motivation für sein GNU-projekt („GNU's Not Unix") an:

> Ich glaube, daß es das Gebot der Nächstenliebe verlangt, daß ich ein Programm, das mir gefällt, mit anderen teile, denen es ebenfalls gefällt. Software-Anbieter hingegen wollen die Anwender isolieren und beherrschen, wobei sie jeden Anwender dazu verpflichten, nicht mit anderen zu teilen. Ich weigere mich, die Solidarität mit anderen Anwendern in dieser Weise zu brechen. Ich kann nicht mit gutem Gewissen einen Nichtoffenbarungsvertrag oder einen Software-Lizenzvertrag unterzeichnen.
>
> Damit ich ehrlich bleiben und trotzdem weiterhin Computer benutzen kann, habe ich mich entschlossen, eine genügend große Sammlung von freier Software zusammenzustellen, so daß ich in der Lage sein werde, ohne jegliche nicht-freie Software auszukommen. Ich habe meinen Beruf im *AI lab* aufgegeben, um dem *MIT* keinen rechtlichen Vorwand zu bieten, mich daran zu hindern, GNU weiterzugeben.

[40] Deutsche Übersetzung in `http://www.gnu.de/documents/manifesto.de.html`

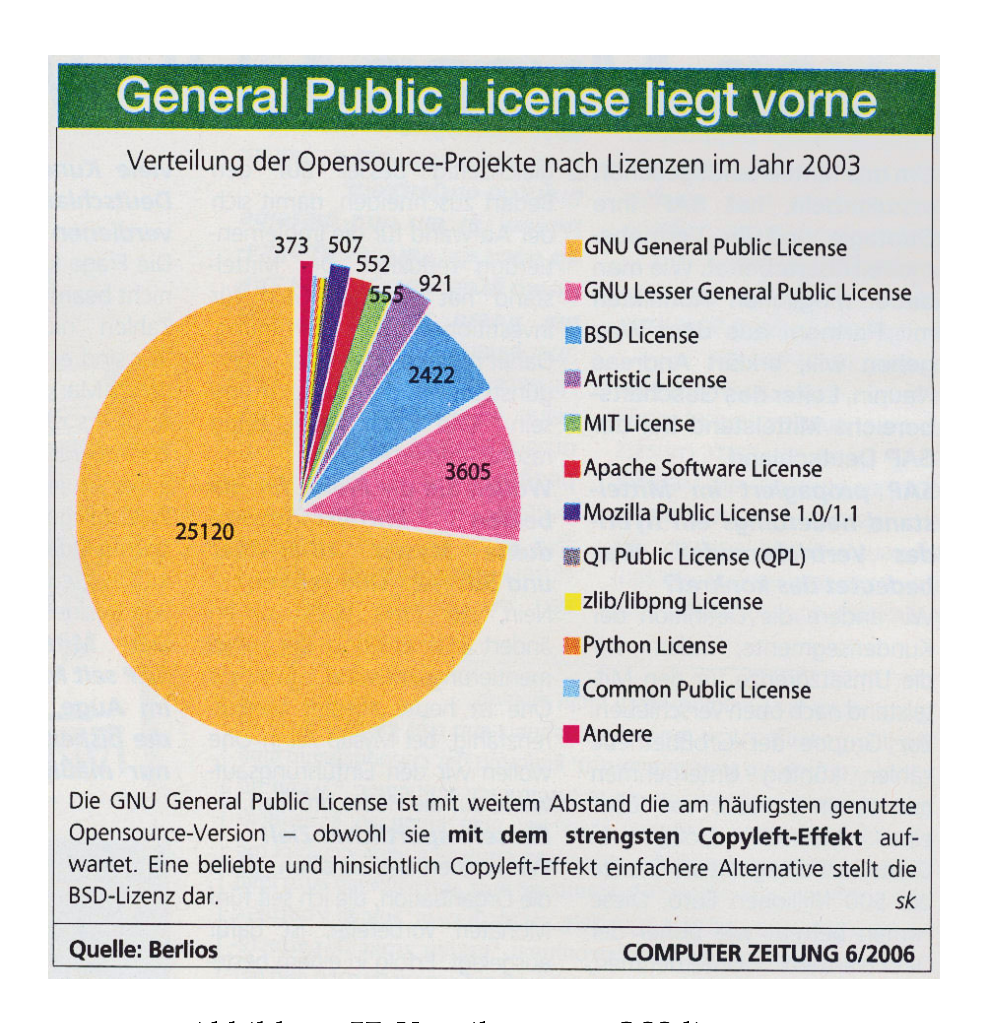


Abbildung 57: Verteilung von OSS-lizenzen

Das lizenzmodell, das Stallman 1984 für seinen *GNU Emacs* formulierte, ist inzwischen in überarbeiteter form als *GPL (GNU General Public Licence)* bekannt und das am weitesten verbreitete modell für open source software (s. abb. 57).

Die freiheiten, die jeder erwerber von software (unabhängig von der frage, ob er für den erwerb etwas zahlen musste oder nicht) besitzt, sind laut Stallman:

1. Die freiheit, das programm zu jedem beliebigen zweck laufen zu lassen.
2. Die freiheit, zu studieren, wie das programm funktioniert, und es an die persönlichen bedürfnisse anzupassen.
3. Die freiheit, kopien weiterzuverteilen, um dem nachbarn zu helfen.
4. Die freiheit, das programm zu verbessern, und die verbesserungen zu publizieren, so daß die ganze gesellschaft davon profitieren kann.

Dabei ist offensichtlich, daß die zweite und vierte dieser freiheiten voraussetzen, daß der quelltext des programms verfügbar ist. Im wortspielerischen gegensatz zum urheberrecht, englisch: *Copyright*, spricht Stallman vom *Copyleft* („left“ heisst nicht nur „links“ – was durchaus zur politischen haltung der hacker im allgemeinen passt – sondern auch „hinterlassen“) und die standardangabe „Alle Rechte vorbehalten — *all rights reserved*“ verwandelt er in *all rights reversed*, alle rechte umgekehrt. Hacker wie Stallman lieben wortspiele. Dazu passt auch das geänderte zeichen (abb. 58).



Abbildung 58: Copyleft

Der trick, mit dem Stallman die verbreitung von freier software fördert, ist nämlich der: jeder, der unter der GPL software annimmt und verändert, verpflichtet sich, sie entweder für sich zu behalten oder aber, wenn er sie weitergeben will, unter der gleichen GPL weiterzugeben und solcherart auch den nächsten in der kette zu gleichem handeln zu verpflichten. Die GPL wirkt damit wie hefe oder sauerteig[41]: eine einzige verwendete zeile aus freier software im sinne der GPL verpflichtet einen verwender, sein ganzes paket entweder still für sich zu behalten oder aber ebenfalls als freie software weiterzugeben. (Es gibt noch eine abgeschwächte version für die verwendung in programmbibliotheken, die *Lesser GPL*.)

An dieser stelle sollte bemerkt werden, daß der begriff *Open Source Software* (OSS) für sich allein mehrdeutig ist und mindestens drei verschiedene interpretationen zulässt:

- Von *Free Software* sollte im grunde nur dann gesprochen werden, wenn damit gemeint ist, daß sie unter der GPL vertrieben wird. Das bedeutet aber – wie bereits zuvor bemerkt – nicht, daß dies kostenlos geschehen muß; es darf durchaus auch Geld dafür verlangt werden.
- Die mit dem Unix-Derivat BSD der University of California Berkeley verbundene *BSD-lizenz* – die nach der GPL und der LGPL dritthäufigst verwendete lizenz – stellt lediglich klar, dass der hersteller nicht für irgendwelche schäden haftet, die durch die software oder daraus abgeleitete produkte entstehen und verlangt im übrigen, dass die copyright-notiz unverändert erhalten bleibt, zwei punkte, die im grunde selbstverständlich sind und auch in der GPL enthalten sind. Davon abgesehen, lässt die BSD-lizenz dem verwender jede freiheit.

[41] Bill Gates spricht aus begreiflichen gründen von der GPL als einem „virus“, das die software verseucht.

- Der begriff *Public Domain* dagegen sagt überhaupt nichts aus, ausser daß jemand seine quellprogramme der öffentlichkeit zur verfügung stellt, d. h. also auf seine besitzrechte verzichtet. In analogie zum strandgutparagraphen kann jeder, der sie findet, damit tun, was immer ihm gefällt.

Google Codesearch[42] bietet (stand november 2011) die folgenden open-source-lizenzmodelle an:

- Aladdin Public License
- Artistic License
- Apache License
- Apple Public Source License
- BSD license
- Common Public License
- Eclipse Public License
- GNU Affero General Public License
- GNU General Public License
- GNU Lesser General Public License
- Historical Permission Notice and Disclaimer
- IBM Public License
- Lucent Public License
- MIT License
- Mozilla Public License
- NASA Open Source Agreement
- Python Software Foundation License
- Q Public License
- Sleepycat License
- Zope Public License

Einen guten überblick über die eigenschaften der am weitesten verbreiteten lizenzmodelle bietet `choosealicense.com`. Auf der startseite werden drei lizenzmodelle für personen angeboten, die sich nicht viele gedanken machen wollen:

simple and permissive die MIT-Lizenz

concerned about patents die Apache-Lizenz

care about sharing improvements die GPL version 2 oder 3.

Wer es genauer wissen will, findet auf der dort verlinkten folgeseite etwas detailliertere Vergleiche (stand vom 17.7.2013), unterteilt in die kategorien „erforderlich", „erlaubt" und „verboten". Hier wird auch darauf hingewiesen,

[42] `www.google.com/codesearch`

dass manche open-source-communities bestimmte dieser lizenzmodelle bevorzugen. Hundertprozentig durchdacht erscheint der vergleich noch nicht überall; es lohnt sich bestimmt, die seite häufiger anzuschauen:

Erforderlich Hier bedeutet

LC (license and copyright Abdruck von lizenz und copyright
SC (state change) Markierung von änderungen
DS (disclose source) Verpflichtung zur veröffentlichung geänderten codes
LU (library usage) Dieser punkt scheint hier falsch zu sein; er gehört eher unter die abteilung „erlaubt“: bibliotheken dürfen in projekten verwendet werden, die nicht „open source“ sind

	LC	*SC*	*DS*	*LU*
Apache v2	X	X		
GPL v2	X	X	X	
MIT License	X			
Mozilla Public License 2.0	X		X	
LGPL v2.1	X		X	X
BSD 3-clause	X			
Artistic License 2.0	X	X	X	
GPL v3	X	X	X	
LGPL v3	X		X	X
Affero GPL	X	X	X	
Public Domain (Unlicense)				
GitHub no license	X			
Eclipse Public License v1.0	X		X	
BSD 2-clause	X			

Erlaubt Hier bedeutet

PU (private use) Privater gebrauch
CU (commercial use) Kommerzieller gebrauch (sollte privaten gebrauch implizieren, aber wozu dann die unterscheidung?)
MO (modification) Modifikation
DI (distribution) Distribution
SL (sublicensing) Erteilung von sublizenzen
PG (patent grant) Patenterteilung

	PU	*CU*	*MO*	*DI*	*SL*	*PG*
Apache v2		X	X	X	X	X
GPL v2		X	X	X		X
MIT License		X	X	X	X	
Mozilla Public License 2.0		X	X	X	X	X
LGPL v2.1		X	X	X	X	X
BSD 3-clause		X	X	X	X	
Artistic License 2.0		X	X	X	X	X
GPL v3		X	X	X		X
LGPL v3		X	X	X	X	X
Affero GPL		X	X	X		
Public Domain (Unlicense)	X	X	X	X	X	
GitHub no license	X	X				
Eclipse Public License v1.0		X	X	X	X	X
BSD 2-clause		X	X	X	X	

Verboten Hier bedeutet

HL (hold liable) Haftung
UT (use trademark) Namen, logos oder warenzeichen dürfen nicht verwendet werden
SL (sublicensing) Erteilung von unterlizenzen
MO (modification)
DI (distribution) Distribution

	HL	*UT*	*SL*	*MO*	*DI*
Apache v2	X	X			
GPL v2	X		X		
MIT License	X				
Mozilla Public License 2.0	X	X			
LGPL v2.1	X				
BSD 3-clause	X	X			
Artistic License 2.0	X	X			
GPL v3	X		X		
LGPL v3	X				
Affero GPL	X		X		
Public Domain (Unlicense)	X				
GitHub no license			X	X	X
Eclipse Public License v1.0	X				
BSD 2-clause	X				

Ob alle diese verschiedenen lizenzmodelle einen irgendwie gearteten sinn ergeben oder in dieser vielfalt wirklich nötig sind, mag dahingestellt bleiben.

Die GPL ist wie alle lizenzvereinbarungen in einem für normale menschen schwer verständlichen juristenchinesisch abgefasst („legalese“); nicht alle passagen erschliessen sich in ihrer bedeutung unmittelbar. Die momentan aktuelle version 3 der GPL ist am 29. Juni 2007 verabschiedet worden. Wirklich verbindlich ist nur der englische text (zu finden unter `http://www.gnu.org/licenses/gpl.html`), aber es gibt aber zur erleichterung für deutsche leser eine inoffizielle übersetzung unter `http://www.gnu.de/documents/gpl-3.0.de.html`. Hieraus sollen einige wesentliche passagen zitiert werden:

4. **Unveränderte Kopien:** Sie dürfen auf beliebigen Medien unveränderte Kopien des Quelltextes des Programms, wie sie ihn erhalten, übertragen, sofern Sie auf deutliche und angemessene Weise auf jeder Kopie einen angemessenen Urheberrechts-Vermerk veröffentlichen, alle Hinweise intakt lassen, daß diese Lizenz und sämtliche gemäß §7 hinzugefügten Einschränkungen auf den Quelltext anwendbar sind, alle Hinweise auf das Nichtvorhandensein einer Garantie intakt lassen und allen Empfängern gemeinsam mit dem Programm ein Exemplar dieser Lizenz zukommen lassen.

   Sie dürfen für jede übertragene Kopie ein Entgelt – oder auch kein Entgelt – verlangen, und Sie dürfen Kundendienst- oder Garantieleistungen gegen Entgelt anbieten.

5. **Übertragung modifizierter Quelltextversionen:** Sie dürfen ein auf dem Programm basierendes Werk oder die nötigen Modifikationen, um es aus dem Programm zu generieren, kopieren und übertragen in Form von Quelltext unter den Bestimmungen von §4, vorausgesetzt, daß Sie zusätzlich alle im folgenden genannten Bedingungen erfüllen:

   a) Das veränderte Werk muß auffällige Vermerke tragen, die besagen, daß Sie es modifiziert haben, und die ein darauf bezogenes Datum angeben.
   b) Das veränderte Werk muß auffällige Vermerke tragen, die besagen, daß es unter dieser Lizenz einschließlich der gemäß §7 hinzugefügten Bedingungen herausgegeben wird. Diese Anforderung wandelt die Anforderung aus §4 ab, „alle Hinweise intakt zu lassen“.
   c) Sie müssen das Gesamtwerk als Ganzes gemäß dieser Lizenz an jeden lizensieren, der in den Besitz einer Kopie gelangt. Diese Lizenz wird daher – ggf. einschließlich zusätzlicher Bedingungen gemäß §7 – für das Werk als Ganzes und alle seine Teile gelten, unabhängig davon, wie diese zusammengepackt werden. Diese Lizenz erteilt keine Erlaubnis, das Werk in irgendeiner anderen Weise zu lizensieren, setzt aber eine derartige Erlaubnis nicht außer Kraft, wenn Sie sie diese gesondert erhalten haben.

d) Wenn das Werk über interaktive Benutzerschnittstellen verfügt, müssen diese jeweils angemessene rechtliche Hinweise anzeigen. Wenn allerdings das Programm interaktive Benutzerschnittstellen hat, die keine angemessenen rechtlichen Hinweise anzeigen, braucht Ihr Werk nicht dafür zu sorgen, daß sie dies tun.

Als durchführungshilfe wird vorgeschlagen, dass jeder unter der GPL publizierte quelltext zu beginn einen kommentar der folgenden form beinhaltet:

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Free/Libre Open Source Software gewinnt weltweit immer mehr freunde, auch und sogar in der industrie, die lange zeit diesem sektor kritisch gegenüberstand, weil jede form von haftung für schäden von allen lizenzmodellen ausgeschlossen wird und deshalb der produzent eines produkts, das FLOSS software enthält, möglicherweise selber in die haftung gerät. Inzwischen sind jedoch zahlreiche sogar sicherheitsrelevante softwarepakete unter verwendung von FLOSS software (mit unterschiedlichen lizenzmodellen) entstanden. Als beispiel kann man etwa die seite `www.mercedes-benz.com/opensource` aufsuchen, welche die in Mercedes-fahrzeugen verwendete FLOSS software mit den jeweiligen lizenzen auflistet. Darin ist die folgende passage enthalten:

> Components of the software used in the vehicle may be free and open source software licensed under the terms of license of the GNU General Public License, Version 2 (GPL) or GNU Lesser General Public License, Version 2.1 (LGPL). Upon request, we will supply the source code of the components licensed under the GPL and LGPL on a data medium (please specify the designation of your vehicle). Please

direct your request to the following address within three years after vehicle delivery:

Daimler AG, HPC: CAC, Customer Service, D-70546 Stuttgart

The copyright holders usually do not provide any warranty and assume no liability whatsoever for the free and open source software components. Note that any modification to the vehicle of any kind can void any warranty claim.



Abbildung 59: Intuitive bedienung

A Die bedeutung der benutzungsschnittstelle

Ein wichtiges thema für die softwaretechnik, wo sehr viele dramatische fehler begangen werden, ist die anlage einer vernünftigen benutzungsschnittstelle, oft auch leicht fälschlich als „benutzerschnittstelle“ bezeichnet. Fehler können hier in der regel vermieden werden, indem benutzer frühzeitig in den entwurfsprozess integriert werden und ihr umgang mit versionen des systems systematisch studiert wird.

Sehr wichtig in diesem zusammenhang ist es natürlich, normen und konventionen einzuhalten, z. B. die vorschriften des CUA (Common User Access), der seit 1987 von IBM im Rahmen der SAA (Systems Application Architecture) festgeschrieben wurde und weithin akzeptiert ist. Laut CUA kann mit einem druck auf die taste `F1` eine kontextsensitive hilfe angefordert werden. Das hinderte einen *professional editor*, den ich selbst einmal auf dem IMB PC im grunde gerne verwendet habe, aber nicht daran, an `F1` die funktion „quit immediately, dont’ ask questions“ zu binden. Solche fehler in der benutzung werden aber nicht nur im softwarebereich, sondern auch sonst in der technik gemacht. Der VW-Porsche (abb. 60) beispielsweise hatte eine 5-gang-schaltung, bei der der rückwärtsgang an der stelle untergebracht war, wo man normalerweise den ersten gang erwartet hätte.

Eine lesenswerte quelle zum thema stellt das ausgezeichnete Buch von Norman (1988) dar. Hier gibt es viele beispiele dafür, dass gegenstände aller art möglichst intuitiv bedienbar sein sollten; die bedienung sollte selbsterklärend sein, wenn das irgendwie möglich ist. Als besonders rühmliches beispiel wird die elektrische sitzverstellung in den Mercedes-fahrzeugen hervorgehoben (s. abb. 59): Was immer man mit dem sitz und der kopfstütze anstellen möchte, braucht man einfach nur mit den als modell des sitzes angelegten schaltern



Abbildung 60: Verstoss gegen normen



Abbildung 61: Falsche benutzerführung

zu tun. Ein gegenbeispiel findet sich in abb. 61: dieser geldautomat suggeriert durch die roten pfeile eine völlig falsche stelle, an der man angeblich seine EC-karte einführen soll, die richtige stelle ist weiter unten und viel unscheinbarer.

Dass die im folgenden angeführten beispiele für absolut missglückte benutzungsschnittstellen beide aus dem hause Siemens stammen, ist fast ein reiner zufall; ich habe diese texte einfach so im abstand von kaum einer woche gefunden und sie haben mir sehr aus der seele gesprochen, weil ich am arbeitsplatz auch mit so einem praktisch unbedienbaren Siemens-telefon geplagt bin. Andererseits steckt vielleicht auch mehr dahinter; möglicherweise haben wir hier ja den eigentlichen grund, warum sich produkte dieser firma nur unter zahlung erheblicher schmiergelder in den markt drängen liessen. Die entscheidung überlasse ich der leserin. . .

> *Siemens ist nicht die Mafia. Man muss das vielleicht so deutlich sagen, um Missverständnisse zu vermeiden.*
> Marc Beise, „Siemens' langer Schatten", Süddeutsche Zeitung Nr. 88, 15. April 2008

A.1 Das Prinzip Siemens

Tobias Kniebe, Süddeutsche Zeitung Magazin, Heft 42, 20.10.2006

Doch, ja, am Ende wird es auch hier um raffgierige Manager, heimtückische Globalisierungspläne und die Vernichtung heimischer Arbeitsplätze gehen. Aber nicht sofort. Zunächst muss vom Handy meiner Mutter die Rede sein. Meine Mutter ist Jahrgang 1944 und wie viele Menschen ihrer Generation steht sie dem Fortschritt mit gesunder Distanz gegenüber. Dass sie überhaupt ein Mobiltelefon hat, hängt hauptsächlich mit dem Wunsch ihrer Kinder zusammen, ihre Bewegungen durch die Welt halbwegs zu verfolgen. Sie ist das, was Handyhersteller gern einen „Einsteiger" nennen. Folgerichtig benutzt sie ein schon etwas älteres, silberfarbenes Einsteigermodell namens A65, und das wurde – jetzt kommt es – von Siemens hergestellt, als Siemens noch Siemens hieß und nicht BenQ und noch keine Rede war von Insolvenz. Wenn meine Mutter ihr Handy nicht versteht, stellt sie gern Einsteigerfragen. Neulich war sie zu Besuch und behauptete, dass es unmöglich sei, Einträge aus ihrem Handy-Adressbuch zu löschen. Das haben wir gleich, sagte ich, und machte mich an ihrem Menü zu schaffen. Es war ohne Zweifel das hässlichste und umständlichste Handymenü, das ich je gesehen hatte. Alles funktionierte genau andersherum, als man spontan erwartet hätte. Und: Ich kam nicht zum Ziel. Ich konnte einzelne Zahlen löschen und einzelne Buchstaben, aber einen ganzen Eintrag? Keine Chance. Ist nicht so wichtig, sagte meine Mutter, die merkte, wie Besessenheit in mir aufstieg. Aber mein Ruf als Gadget-Guru stand auf dem Spiel. Ich lud mir die Bedienungsanleitung von der früheren

Siemens- und jetzigen BenQ-Website herunter; ich konsultierte Internetforen. Die Stunden vergingen. Nichts half. Am Ende, als meine Mutter längst im Bett war, gab ich auf, legte ihre Karte in mein Nokia ein, löschte, änderte und speicherte ihre Daten innerhalb von drei Minuten – und fiel danach in einen unruhigen Traum, in dem der Technikstandort Deutschland sehr düster, karstig und menschenleer aussah. Ungefähr so wie das Land Mordor im Herrn der Ringe. Das Handy meiner Mutter, ich habe nachgeschaut, kam im Herbst 2004 auf den weltweit expandierenden Markt der Mobiltelefone. Im Sommer 2005 war Siemens vom dritten Platz der Handyhersteller bereits auf den fünften Platz zurückgefallen, der Marktanteil auf 5,5 Prozent abgesackt. Seither ging es weiter bergab. Der Zusammenhang ist – zumindest mir – nun sonnenklar. Politiker, Gewerkschaftsfunktionäre und Wirtschaftskommentatoren, die sich über den großen Niedergang noch wundern oder erregen können, haben wahrscheinlich schon lange kein Siemens-Handy mehr benutzt, und ganz sicher haben sie nie versucht, einen überflüssigen Adressbucheintrag wie beispielsweise den „Vodafone Blumengruss“ aus ihrem Siemens-Handy zu löschen. Aber Moment: Kann das alles wirklich so simpel sein? Ich fürchte: ja. In der Welt der Konsumgüter liegen die Beweise schließlich für jeden auf der Hand. In diesen Wochen ist nun viel von der Verantwortung für die Mitarbeiter die Rede, von Werten wie Stabilität, Rationalität und Kontinuität, die traditionell mit dem Prinzip Siemens verbunden waren und es nun wohl nicht mehr sind. Dass diese Werte jedoch früher auch auf einem Handwerksethos beruhten, das jedem einzelnen Siemens-Mitarbeiter wichtig war; dass in der Siemenskultur keiner etwas aus der Hand gab, was nicht gut durchdacht und technisch ausgereift war – davon spricht merkwürdigerweise niemand. Die Siemens-Topmanager in ihrer Gier und ihrer strategischen Unfähigkeit mögen an vielem schuld sein, aber die Menü-Software im Handy meiner Mutter haben sie nicht entwickelt. Wer auch immer dafür die Verantwortung trägt, wer das getestet, abgesegnet oder versucht hat, dieses Handy zu verkaufen, ohne einen internen Aufstand anzuzetteln – der muss jetzt auch der anderen Seite des Desasters ins Auge sehen: Kein Politiker, kein Gewerkschafter, kein Sozialplan kann uns auf Dauer vor den Folgen retten, wenn unsere Arbeit derart katastrophale Ergebnisse produziert.

A.2 LEBENSZEICHEN

Sie haben Nachrichten

Martenstein kämpft mit seinem Anrufbeantworter

Harald Martenstein, Die ZEIT Nr. 44, 26. Oktober 2006

Trotz all dieser Probleme, die ich in der letzten Zeit hatte, möchte ich um Gottes Willen nicht den Eindruck erwecken, ich sei verbittert. Ich bin nicht

verbittert. Ich bin gut drauf. Sicher, im Detail könnte manches besser laufen. Als ich nach Hause kam, blinkte an dem neuen Telefon ein roter Knopf. Auf dem Display stand: „Sie haben neue Nachrichten." Bei meinem alten Telefon wurden, wenn man auf den roten Knopf drückte, die neuen Nachrichten vorgespielt. Hier aber leuchteten stattdessen zwei Botschaften auf dem Display auf, nämlich „Neue Nachr.: 1/2" sowie das Wort „Pause". Ich dachte, sehr wahrscheinlich habe ich zwei neue Nachrichten. Jetzt fragt mich das Telefon, ob ich lieber die erste oder lieber die zweite der beiden neuen Nachrichten hören oder ob ich lieber eine Pause machen möchte. Ich dachte, jemand, der nach Hause kommt, möchte doch mit 99,9-prozentiger Wahrscheinlichkeit ohne weitere Umschweife hintereinander seine neuen Nachrichten hören, zack, zack, und nicht, nachdem er erfahren hat, dass es zwei Nachrichten gibt, erst einmal eine Auszeit nehmen, um diese Information geistig zu verarbeiten. Die Leute, die dieses Telefon gebaut haben, wissen allzu wenig über die menschliche Psychologie. Ich habe den Knopf „Weiter" gedrückt. Daraufhin bot mir das Telefon die folgenden Möglichkeiten an: „Weiter hören", „Akt. Nachr. löschen", „Nummer wählen", „Kurzwahlliste", „Wiederholen", „Alte Nachr. löschen", „Anzeigen", „Auf ‚neu' setzen", „Nr. ins Tel.buch", „Geschwindigkeit" und „Beenden". Ich dachte, dieses Telefon ist ja eine Metapher auf die Kapitalismuskritik. Auswahl gibt es, gewiss, nur das, was der Mensch wirklich braucht, ist nicht dabei, nämlich „dir endlich deine neuen Nachrichten vorspielen, Brüderchen". Ich habe „Weiter hören" gewählt. Der Anrufbeantworter spielte Nachricht Nummer zwei vor. Das hat mir nicht genügt. Wir wollen alles. In der Hoffnung, die erste Nachricht zu erfahren, habe ich nochmals den Knopf „Weiter" gedrückt. Jetzt erschienen auf dem Display fünf Bildchen, es waren eine Musiknote, eine Aktentasche, eine Uhr, ein Schraubenschlüssel und eine Art Fernseher. Also wählte ich, aus Neugierde, den Fernseher. Jetzt erschienen, potz Blitz!, die Worte „Anrufbeantwort." und „Anruferliste" auf dem Display. Klar, ich habe „Anrufbeantwort." gedrückt. Das Telefon spielte nun aber keineswegs Nachricht eins ab, sondern bot wieder vier Möglichkeiten an: „Nachrichten", „Infos", „Ansagen" und „AB Ausschalten?". Ich wählte „Nachrichten", worauf das Telefon anbot: „Neue Nachrichten" und „Alle Nachrichten". Da dachte ich, fuck you, you phone of a bitch. Ich habe die Tagesschau geguckt. Da kommen auch Nachrichten. Vielleicht sollte ich erwähnen, dass es ein Telefon von Siemens ist. Nein, verbittert bin ich überhaupt nicht. Dies alles werde ich eines Tages in den Griff bekommen.

B Nichtfunktionale eigenschaften von software

Jede software hat natürlich gewisse *funktionale eigenschaften*; das sind eigenschaften, die sich auf die funktion der software beziehen: eine textverarbeitungssoftware soll text verarbeiten, eine tabellenkalkulation tabellen kalkulieren usf. Darüber hinaus gibt es jedoch auch eine menge in der literatur dokumentierter allgemeinerer anforderungen an industriell verwendbare software, die von der beabsichtigten funktion unabhängig sind und sich eher auf die gebrauchseigenschaften beziehen. Sie heissen deshalb *nichtfunktionale anforderungen*; viele davon lassen sich im weitesten sinne unter der überschrift „produktqualität" zusammenfassen.



Abbildung 62: Qualitätsbäume

Ich habe die meiner meinung nach wichtigsten 18 nichtfunktionalen anforderungen ausgewählt und versucht, in ein diagramm (abb. 62) so einzuordnen, dass gewisse anforderungen als spezialfälle bzw. komponenten von allgemeineren anforderungen gesehen werden. Über diese einordnung („taxonomie") lässt sich natürlich trefflich streiten. Es gibt „konkurrenzmodelle" von

Ludewig (abb. 63), McCall (abb. 64), der ISO (British Standards Institute 2011) und zahlreiche andere.

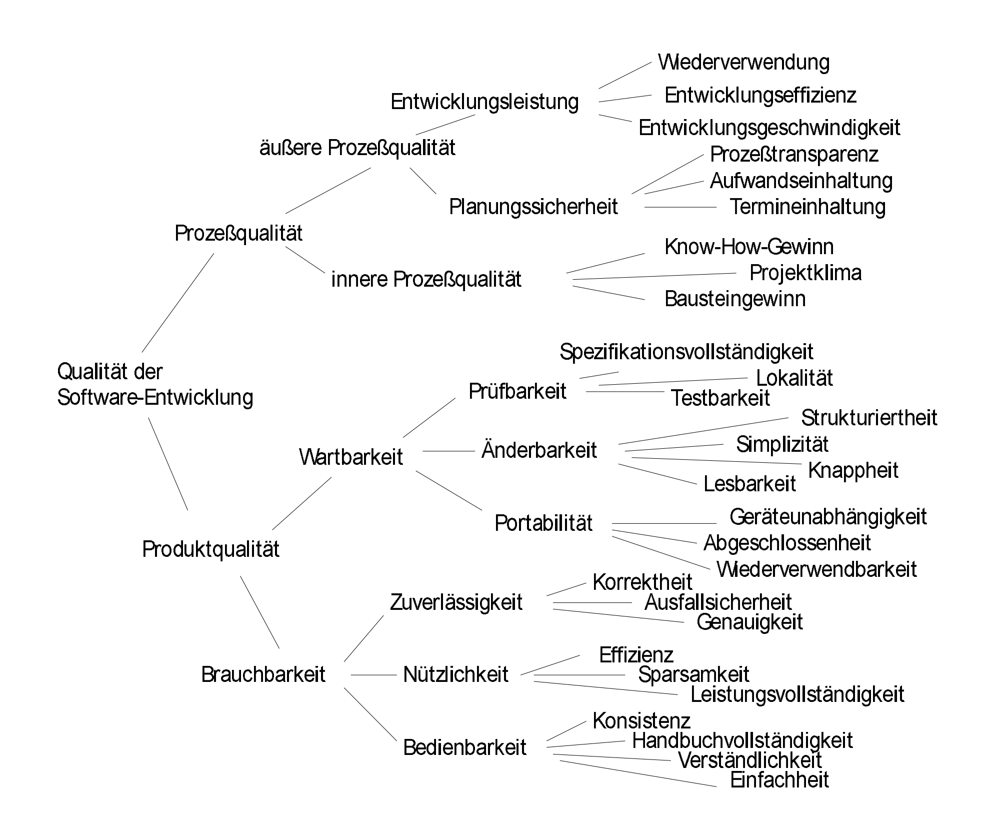


Abbildung 63: Qualitätsbaum nach Ludewig

Das besondere an den diagrammen von McCall ist, dass sie nicht wie die anderen ansätze taxonomien beschreiben, bei denen sich gewisse eigenschaften als bestandteile anderer eigenschaften darstellen, sondern statt dessen darstellen, welche eigenschaften eine auswirkung auf andere eigenschaften haben.

Bei der ISO (British Standards Institute 2011) wird zwischen *qualität im gebrauch (quality in use)* und *system/-software-produktqualität* unterschieden.

Klar sein muss dabei, dass die meisten dieser eigenschaften sich gegenseitig beeinträchtigen und behindern; wir können nicht alle diese ziele gleichzeitig erreichen. Das wurde auch schon im einleitungskapitel anhand des zitats über die widersprüchlichen ziele deutlich, denen sich die ingenieurtätigkeit unterwerfen muss. Karen Mackey (1996) von der Lotus Development Corporati-

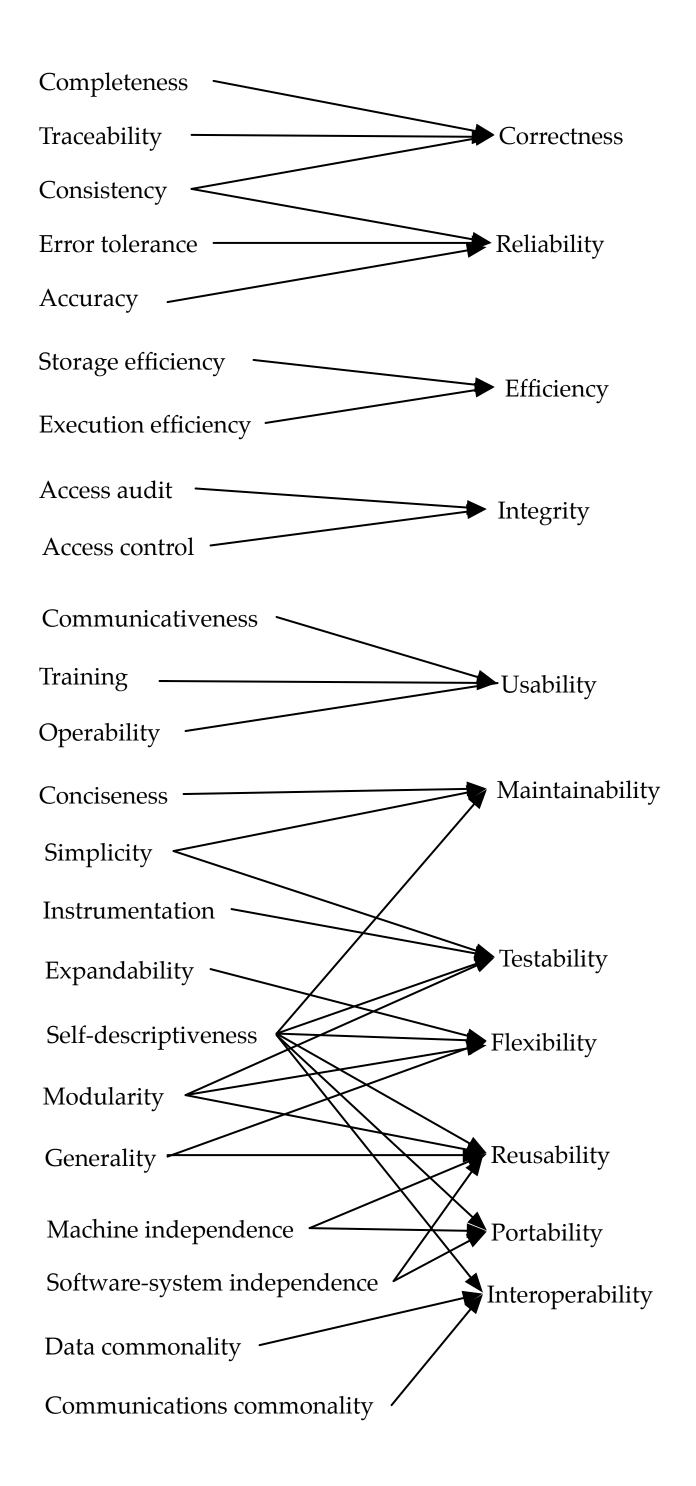


Abbildung 64: Qualitätsdiagramme nach McCall

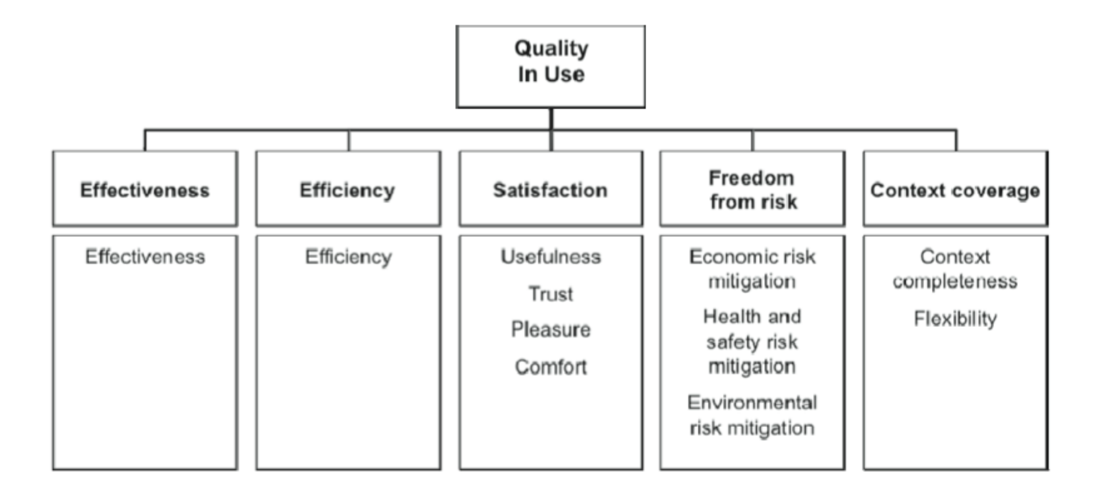


Figure 3 — Quality in use model

Abbildung 65: Qualität im gebrauch nach ISO

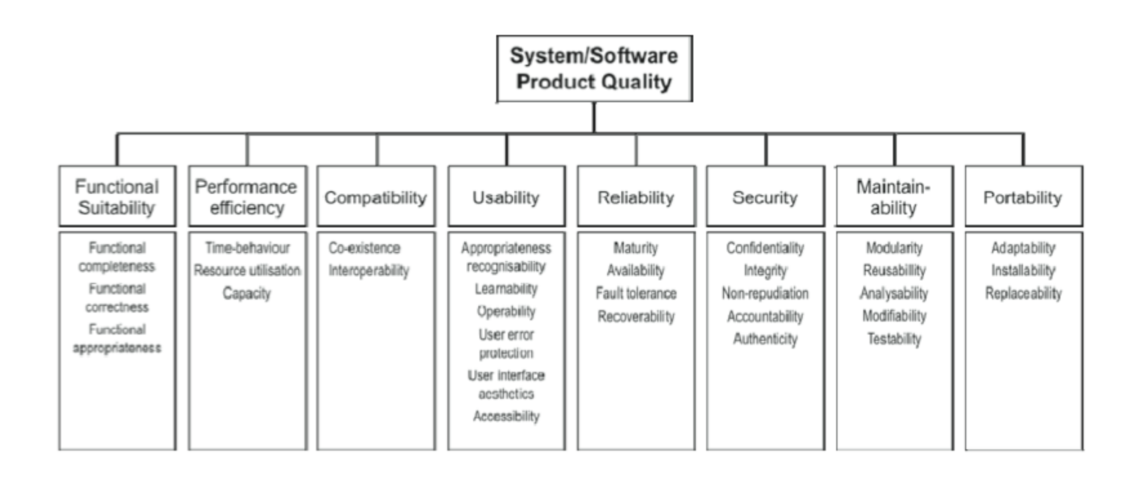


Figure 4 — Product quality model

Abbildung 66: Produktqualität nach ISO

on führt etwa das „Quality-Capacity Syndrome“ als eine der beiden hauptursachen für das scheitern an sich guter projekte an[43]. Hier wird während der entwicklung zwar grosser wert auf die qualität der software im sinne der korrektheit, wartbarkeit etc. gelegt, aber die notwendigen leistungsmerkmale nicht genügend berücksichtigt. Beim einsatz unter realistischen bedingungen stellt sich dann heraus, dass grundlegende änderungen am entwurf nötig sind (s. das zitat von Mackey im anhang). Komplexe systeme fallen häufig gerade dann aus, wenn sie am dringendsten benötigt werden. Beispielshalber fiel das fahrkartenverkaufs- und -reservierungssystem der amerikanischen eisenbahnen (Amtrak) ende November 1996 gerade an dem wochenende aus, an dem normalerweise der reiseverkehr am stärksten ist. Bei starker belastung (hoher füllungsgrad von puffern, häufiges rücksetzen aufgrund von fehlertoleranzmassnahmen) geraten systeme in ungetestete betriebszustände. Irgendwann geht dann gar nichts mehr.

Damit wir über qualität diszipliniert reden können, muss sie *messbar* gemacht werden und ausserdem bestimmte *qualitätssicherungsverfahren* als kontrolle eingeführt werden; siehe hierzu abschnitt 5. In diesem zusammenhang ist auch lesenswert das papier von Boehm und In (1996) über die *quality-requirement conflicts*.

Es folgen, alphabetisch sortiert, definitionen für die von der ISO verwendeten begriffe.

Analysierbarkeit (*analysability*) Grad der effektivität und effizienz, zu dem es möglich ist, die wirkung einer beabsichtigten änderung eines oder mehrerer teile eines produkts oder systems zu beurteilen oder ein produkt auf fehler oder fehlerursachen zu untersuchen oder die zu modifizierenden teile zu bestimmen.

Ästhetik der benutzungsschnittstelle (*user interface aesthetics*) Grad, zu dem eine benutzungsschnittstelle angenehme und befriedigende interaktion für den benutzer ermöglicht.

Anpassbarkeit (*adaptability*) Grad, zu dem ein produkt oder system effektiv und effizient an unterschiedliche oder fortentwickelte hardware-, software- oder andere betriebs- oder einsatzumgebungen angepasst werden kann.

Befriedigung (*satisfaction*) Der grad, zu dem bedürfnisse der benutzer befriedigt werden, wenn das produkt oder system im spezifizierten gebrauchskontext eingesetzt wird.

[43]Das andere ist das fehlen geeigneter werkzeuge (missing tools crisis)

Brauchbarkeit (*usability*) Grad, zu dem ein produkt oder system von spezifizierten benutzern gebraucht werden kann, um spezifizierte ziele mit effektivität, effizienz und befriedigung in einem spezifizierten einsatzkontext zu erreichen.

Effektivität (*effectiveness*) Genauigkeit und vollständigkeit mit dem die benutzer spezifizierte ziele erreichen.

Effizienz (*efficiency*) Abwägung des aufwands im vergleich zur genauigkeit und vollständigkeit, mit der benutzer ziele erreichen.

Erkennbarkeit der Angemessenheit (*appropriateness recognizability*) Grad, zu dem benutzer erkennen können, ob ein produkt oder system für ihre bedürfnisse angemessen ist.

Erlernbarkeit (*learnability*) Etwas kompliziert formuliert: Grad, zu dem ein produkt oder system von spezifizierten benutzern benutzt werden kann, um spezifizierte ziele zu erreichen, wie sie lernen, das produkt oder system mit effektivität, effizienz, risikofreiheit und befriedigung in einem spezifizierten einsatzkontext nutzen können.

Ersetzbarkeit (*replaceability*) Grad, zu dem ein produkt ein anderes spezifiziertes softwareproduct zum gleichen zweck in der gleichen umgebung ersetzen kann.

Fehlertoleranz (*fault tolerance*) Grad, zu dem ein system, produkt oder eine komponente wie geplant funktioniert trotz der gegenwart von hardware- oder softwarefehlern.

Flexibilität (*flexibility*) Grad, zu dem ein produkt oder system mit effektivität, effizienz, risikofreiheit und befriedigung über die anfänglich in den anforderungen spezifizierten einsatzkontexte hinaus benutzt werden kann

Funktionale angemessenheit (*functional appropriateness*) Grad, zu dem ein produkt oder system die erfüllung spezifizierter aufgaben und ziele erleichtert

Funktionale eignung (*functional suitability*) Grad, zu dem ein produkt oder system funktionen bereitstellt, die ausgedrückte und implizierte bedürfnisse befriedigen, wenn sie unter den spezifizierten bedingungen gebraucht werden.

Funktionale korrektheit (*functional correctness*) Grad, zu dem ein produkt oder system die korrekten resultate im notwendigen präzisionsgrad liefert

Funktionale vollständigkeit (*functional completeness*) Grad, zu dem die funktionen alle spezifizierten ziele und zielvorgaben der benutzer erfüllen

Glaubwürdigkeit (*authenticity*) Grad, zu dem bewiesen werden kann, dass die identität eines gegenstands oder einer ressource mit der behaupteten übereinstimmt.

Integrität (*integrity*) Grad, zu dem ein system, produkt oder eine komponente unberechtigten zugriff auf oder modifikation von computerprogrammen oder daten verhindert.

Installierbarkeit (*installability*) Grad der effektivität und effizienz, zu dem ein produkt oder system erfolgreich in einer spezifizierten umgebung installiert oder deinstalliert werden kann.

Interoperabilität (*interoperability*) Grad, zu dem zwei oder mehr systeme, produkte oder komponenten informationen austauschen und die ausgetauschte information benutzen können.

Kapazität (*capacity*) Grad, zu dem die obergrenzen eines produkt- oder systemparameters den anforderungen entsprechen

Koexistenz (*coexistence*) Grad, zu dem ein produkt oder system seine geforderten funktionen effizient ausführen kann, während es eine gemeinsame umgebung und ressourcen mit anderen produkten teilt, ohne schädliche auswirkungen auf irgendein anderes produkt.

Komfort (*comfort*) Zufriedenheitsgrad der benutzer mit dem physischen Komfort

Kompatibilität (*compatibility*) Grad, zu dem ein produkt, system oder eine komponente informationen mit anderen produkten, systemen oder komponenten austauschen kann und/oder seine geforderten funktionen ausführen kann, indem es sich die gleiche hardware- oder softwareumgebung teilt.

Kontextabdeckung (*context coverage*) Grad, zu dem ein produkt oder system mit effektivität, effizienz, risikofreiheit und befriedigung benutzt werden kann, sowohl in spezifizierten einsatzkontexten als auch in kontexten über die anfänglich ausdrücklich identifizierten hinaus.

Kontextvollständigkeit (*context completeness*) Grad, zu dem ein produkt oder system mit effektivität, effizienz, risikofreiheit und befriedigung in allen spezifizierten einsatzkontexten benutzt werden kann

Modifizierbarkeit (*modifiability*) Grad, zu dem ein produkt oder system effektiv und effizient modifiziert werden kann, ohne fehler einzuführen oder die existierende produktqualität zu vermindern.

Modularität (*modularity*) Grad, zu dem ein system oder computerprogramm aus diskreten komponenten aufgebaut ist, so dass eine änderung in einer komponente minimale auswirkungen auf andere komponenten hat.

Nachweisbarkeit (*non-repudiation*) Grad, zu dem nachgewiesen werden kann, dass aktionen oder ereignisse stattgefunden haben, so dass ereignisse und aktionen später nicht abgestritten werden können.

Operabilität (*operability*) Grad, zu dem ein produkt oder system attribute hat, die es leicht machen, es zu betreiben und zu steuern.

Nützlichkeit (*usefulness*) Grad, zu dem benutzer zufriedengestellt werden in ihrer gefühlten erreichung pragmatischer ziele einschliesslich der resultate und konsequenzen des gebrauchs.

Performanzeffizienz (*performance efficiency*, ein merkwürdiger begriff?) Performanz relativ zu den ressourcen unter den angegebenen bedingungen

Portabilität (*portability*) Grad der effektivität und effizienz, zu dem ein system, produkt oder eine komponente von einer hardware-, software- oder anderen betriebs- oder einsatzumgebung auf eine andere übertragen werden kann.

Reifegrad (*maturity*) Grad, zu dem ein system, produkt oder eine komponente die bedürfnisse nach zuverlässigkeit unter normalem betrieb erfüllt.

Risikofreiheit (*freedom of risk*) Grad, zu dem ein produkt oder system das potentielle risiko mindert in bezug auf wirtschaftlichen status, menschenleben, gesundheit oder umwelt

Ressourcenverbrauch (*resource utilization*) Grad, zu dem mengen und typen von ressourcen eines produkts oder systems bei der ausführung seiner funktionen den anforderungen entspricht

Schutz vor benutzerfehlern (*user error protection*) Grad, zu dem ein system den benutzer davor schützt, fehler zu machen.

Sicherheit (*security*) Grad, zu dem ein produkt oder system informationen und daten schützt, so dass personen oder andere produkte oder systeme den für ihre typen und stufen von berechtigung angemessenen grad von zugang zu daten haben.

Testbarkeit (*testability*) Grad der effektivität und effizienz, zu dem testkriterien für ein system, produkt oder eine komponente aufgestellt und tests durchgeführt werden können, um festzustellen, ob diese kriterien erfüllt sind.

Verfügbarkeit (*availability*) Grad, zu dem ein system, produkt oder eine komponente betriebsbereit und zugänglich ist, wenn es für den gebrauch erforderlich ist.

Vergnügen (*pleasure*) Grad des vergnügens, das ein benutzer bei der erfüllung seiner persönlichen bedürfnisse empfindet.

Vertrauen (*trust*) Grad der zuversicht eines benutzers oder interessenvertreters dass das produkt oder system sich wie beabsichtigt verhält.

Vertraulichkeit (*confidentiality*) Grad, zu dem ein produkt oder system sicherstellt, dass daten nur für die zugänglich sind, die eine zugriffsberechtigung haben.

Wartbarkeit (*maintainability*) Grad der effektivität und effizienz, mit der ein produkt oder system von dem beabsichtigten wartungspersonal modifiziert werden kann.

Wiederherstellbarkeit (*recoverability*) Grad, zu dem im fall einer unterbrechung oder eines fehlers ein produkt oder system die direkt betroffenen daten wiederherstellen kann und den gewünschten zustand des systems wieder einstellen kann.

Wiederverwendbarkeit (*reusability*) Grad, zu dem ein gut (*asset*) in mehr als einem system oder zum bau anderer güter verwendet werden kann.

Zeitverhalten (*time behaviour*) Grad, zu dem antwort- und verarbeitungszeit sowie durchsatz eines produkts oder systems bei der ausführung seiner funktionen den anforderungen entspricht

Zugänglichkeit (*accessibility*) Grad, zu dem ein produkt oder system von personen mit einem möglichst weiten bereich von eigenschaften und fähigkeiten benutzt werden kann, um ein spezifiziertes ziel in einem spezifizierten einsatzkontext zu erreichen.

Zurechenbarkeit (*accountability*) Grad, zu dem die aktionen einer entität eindeutig auf diese entität zurückgeführt werden können.

Zuverlässigkeit (*reliability*) Grad, zu dem ein system, produkt oder eine komponente spezifizierte funktionen unter spezifizierten bedingungen über einen spezifizierte zeitraum hinweg durchführt.

C Historisches

Hier werden einige informationen zusammengetragen, die jede(r) informatiker(in) eigentlich wissen sollte, auch wenn sie weniger wichtig sind.

C.1 Die softwarekrise

In der anfangszeit der computerentwicklung[44] und -anwendung sah man in der software allgemein kein problem (Metropolis u. a. 1980); die interessanten fragen lagen eindeutig auf dem hardwaresektor. Der computer galt, wie F. L. Bauer (1991) es heute formuliert, als eine art „vollautomatische handkurbelrechenmaschine“. Das drückt sich auch in den damaligen bezeichnungen „rechenautomat“ bzw. „automatische rechenmaschine“ aus. Auch die ersten kommerziell vertriebenen rechner wurden vom hersteller praktisch ohne jede software ausgeliefert, was für heutige verhältnisse unvorstellbar ist.

Dieses vorgehen war nur deshalb möglich, weil die damaligen computer einerseits noch ziemlich überschaubar waren und andererseits auch nur für ganz eng umgrenzte anwendungen (z.b. numerische algorithmen) eingesetzt wurden.

Die charakteristika aus der frühzeit der computer (etwa 1942–1948) sind

- Programme waren nachbildungen von algorithmen, die aus der mathematik bekannt waren.
- Konstrukteure, programmierer und benutzer bildeten eine im wesentlichen geschlossene gruppe mit durchweg gleicher vorbildung.
- Die programme behandelten eine eingeschränkte problemklasse und griffen nicht unmittelbar in irgendwelche handlungsabläufe ein.

In diesem szenarium war natürlich kein bedarf für „software engineering“; die bekannten techniken aus der mathematik bzw. der elektrotechnik (oder gar mechanik) reichten völlig aus[45]. Spätestens in den sechziger jahren wurde es aufgrund der fortschritte auf dem hardwaresektor möglich, systeme zu entwickeln, die so komplex waren, dass sie sich durch blosse umsetzung mathematischer algorithmen in computerprogramme offensichtlich nicht mehr beschreiben liessen. Ein beispiel ist das SAGE-system von 1951 (Everett (1980):

[44]Die folgenden abschnitte sind zitiert aus Klaeren (1994).

[45] Dieselben charakteristika findet man auch heute noch durchweg in der universitären programmierausbildung wieder, was einer der gründe sein mag, warum softwaretechnik so schwer lehrbar ist.

„Semi-automated ground environment") ein früher SDI-urahn. Dieses system bestand aus einem netz von radarstationen, die mit einem rechner verbunden waren, und sollte automatisch vor einem feindlichen fliegerangriff warnen. Es stellte sich an diesem beispiel wie an zahlreichen anderen heraus, dass programmieren eben nicht nur aus der umsetzung bekannter algorithmen in eine computersprache besteht, sondern dass in den meisten fällen algorithmen erst einmal *gefunden* werden müssen. Hierzu ist als wichtigste voraussetzung die *präzise spezifikation* des problems notwendig. Im SAGE-beispiel etwa war zunächst einmal zu klären, was eigentlich ein flugzeug ist, d.h. inwiefern sich seine radarsignale von denen eines vogelschwarms oder anderer flugobjekte unterscheiden. Hat man diese frage gelöst, ist zu klären, wodurch sich ein feindliches flugzeug von einem eigenen unterscheidet etc. Diese fragen sind ausserordentlich schwer zu entscheiden und blieben im SAGE-system ungelöst. Noch 1960 hat ein SAGE-nachfolger, das *Ballistic Missile Early Warning System*[46] den aufgehenden mond als einen feindlichen raketenangriff gemeldet (Licklider 1969, p. 122–123). Dass die problematik bis heute im wesentlichen ungelöst ist, beweisen der abschuss eines britischen gazelle-hubschraubers durch das britische kriegsschiff Cardiff im falklandkrieg (Neumann 1987) und der abschuss eines iranischen verkehrsflugzeugs durch den amerikanischen kreuzer Vincennes im golfkonflikt (Neumann 1988, 1989)[47].

Wir haben zuvor drei wesentliche merkmale der frühzeit der programmierung vorgestellt; stellen wir jetzt dem gegenüber, was danach charakteristisch wurde und bis heute gilt:

- In den meisten fällen ist der computer nur teil einer ganzheitlichen problemlösung; die genaue beschreibung und aufteilung der aufgabe sowie geeignete algorithmen zu ihrer maschinellen lösung müssen erst gesucht werden.
- Maschinenhersteller, programmierer und benutzer der rechner bzw. programme sind verschiedene menschengruppen mit unterschiedlicher vorbildung. Dies hat kommunikationsprobleme zur folge.
- Der einsatzbereich der programme ist unüberschaubar; die programme greifen z.t. unmittelbar in praktische handlungsabläufe ein.

Von F. L. Bauer (1993) wurde deshalb mitte der sechziger jahre im wesentlichen der folgende gedanke vorgetragen: „Wenn es wahr ist, dass die entwick-

[46]Interessanterweise ist die abkürzung dieses systems „BMEWS", was sich auch als „bemuse" (=„irritieren") aussprechen lässt.

[47]Der Vincennes-zwischenfall eröffnet eine neue dimension: Die software hat hier, zumindest was die kursangaben des verkehrsflugzeugs angeht, richtig gearbeitet; es war die komplexität der benutzerschnittstelle mit vier separaten alphanumerischen bildschirmen, die zu dem fehler geführt hat.

lung komplexer softwaresysteme unsystematisch geschieht und wenn andererseits bekannt ist, dass die ingenieurwissenschaften sich traditionell mit der entwicklung komplexer systeme beschäftigen, dann soll man doch versuchen, auch softwaresysteme nach ingenieursmässigen prinzipien und regeln zu erstellen."

Im nachgang zu der berühmten konferenz des NATO Science Committees 1968 in Garmisch-Partenkirchen (Naur und Randell 1969), die von F. L. Bauer vorbereitet und durchgeführt wurde, ist das neue schlagwort „Software Engineering" begeistert aufgenommen worden und hat seitdem seinen festen platz in der informatik eingenommen. Christiane Floyd (1994) sagt:

> *Nach meiner Auffassung hat Software-Engineering die Softwarekrise der 60er Jahre im Prinzip bewältigt. Insbesondere liegen für Modellierung, Softwarearchitektur und Programmierung ausgereifte lehrbare Konzepte vor, die vielfach in der Praxis erprobt sind. Zu bedauern ist, daß nach wie vor in der Praxis großenteils von ihnen keine Kenntnis genommen wird, so daß die alten Probleme noch immer nicht generell überwunden sind.*

An anderer Stelle bemerkt sie dann allerdings

> *Besonders gravierend ist der Widerspruch zwischen dem anerzogenen Selbstverständnis der Softwareingenieure und -ingenieurinnen, sie sollten (nur) algorithmische Lösungen für wohldefinierte Probleme erarbeiten, und den Anforderungen der Praxis an Flexibilität, Kommunikations- und Teamfähigkeit. [...] Diese Probleme halte ich für so gravierend, daß sie angesichts der veränderten wirtschaftlichen Situation zu einer neuen Softwarekrise der 90er Jahre führen könnten.*

C.2 Qualifikation des softwareingenieurs

Das zitat von Floyd zielt auf das selbstverständnis der softwareingenieure. Bereits im einleitungskapitel haben wir die tätigkeit des softwaretechnikers oder softwareingenieurs verglichen mit der eines architekten. In der tat sind eine fülle von gemeinsamkeiten zu entdecken. Wie der architekt, so gestaltet auch der softwaretechniker räume, in denen menschen leben und arbeiten können bzw. müssen. Beim architekten sind die räume real zu verstehen, beim softwaretechniker eher im übertragenen sinne. Beide berufe schaffen jedoch strukturen und umgebungen, die einen wesentlichen einfluss darauf ausüben, ob sich menschen darin und damit wohlfühlen können. Nicht umsonst sprechen wir ja auch von der softwarearchitektur. Schon 1979 hat Heinz Zemanek auf diese verbindung hingewiesen und in bezug auf die softwarearchitektur den leibarchitekten von kaiser Augustus, Lucius Vitruvius Pollio („Vitruv", ca. 84–27 v. chr.), zitiert, der 10 bücher über die architektur geschrieben hat. Das erste kapitel im ersten buch hiervon (*De architectis instituendis*) beschäftigt sich mit der ausbildung der architekten und beginnt mit dem satz: „Die wissenschaft

des architekten ist durch mehrere disziplinen und verschiedenartige ausbildungsbereiche geschmückt[48]; durch sein urteil werden alle werke bewertet, die von den übrigen künsten ausgeführt werden". In der tat wird auch für die softwaretechnik, teilweise sogar für die informatik im allgemeinen der interdisziplinäre charakter immer häufiger betont. Etwas weiter im text fordert Vitruv (s. S. 239) vom architekten:

> *„Deshalb muss er auch erfindungsreich[49] und gelehrig in der ausbildung sein, denn weder einfallsreichtum ohne ausbildung noch ausbildung ohne erfindungsgabe können einen vollkommenen künstler hervorbringen."*

Diese sätze sollte man wahrscheinlich mit goldenen buchstaben über jede ausbildung in softwaretechnik schreiben. Etwas spezifischer sagt Vitruv vor über 2000 jahren über den architekten folgendes:

> *Er soll viel gelesen haben und erfahren sein mit dem bleistift, ausgebildet in geometrie, nicht unkundig der optik, unterwiesen in arithmetik; er wird viel geschichte gelernt haben und die philosophen aufmerksam gehört haben. Er wird musik kennen und nicht unkundig in der medizin sein. Er wird die kommentare der rechtsgelehrten kennen und soll astrologie und die massverhältnisse des himmels kennengelernt haben.*

Zemanek bemerkt dazu, dass dies in der tat ein umfassendes programm ist, dessen grösse fast das fehlen jeglicher systematischen ausbildung für architekten rechtfertigen könnte, macht sich dann aber sehr stark für eine solch umfassende ausbildung, auch für softwaretechniker und computerarchitekten. Die breite ausbildung, die Vitruv für einen architekten fordert, so Zemanek, hat in der tat nicht unbedingt einen direkten einfluss auf die produkte, die der architekt schaffen kann. Vitruv sieht sie eher als vorausbedingung für eine gewisse geisteshaltung an, in der man den entwurf durchführen kann. Mit anderen worten ist es nicht genug, die zugrundeliegende technik zu beherrschen, noch reicht es, die aufgabe verstanden zu haben und sie so gut zu lösen wie eine schmalbandige ausbildung erlauben kann, sondern architekturentwurf muss von höchster qualität sein und alle diesbezüglichen kenntnisse umfassen. Zemanek bemerkt, dass entwurfssünden erbsünden sind:

> *„Sie binden die nachfolgenden Generationen und das ist viel schlimmer als es klingt. Unsere Nanosekunden verführen alle Parteien in den Irrglauben, dass man im Computer und mit Hilfe des Computers mit hoher Geschwindigkeit handeln und korrigieren kann. Es gibt sicherlich superschnelle Prozesse, aber nicht im Bereich des Entwurfs und noch weniger in der Korrektur von Systemkonzepten."*

[48] *Architecti est scientia pluribus disciplinis et variis eruditionibus ornata.* Heute würden wir sagen, dass es sich um eine interdisziplinäre wissenschaft handelt.

[49] lat. *ingeniosus*, daher das wort ingenieur

In ganz ähnlicher art und weise argumentiert MacLennan (1997), welcher die Softwaretechnik mit dem bauingenieurwesen vergleicht und hierzu ein buch von Billington (1985) zitiert. Hiernach hat die technik drei dimensionen: die wissenschaftliche (*scientific*), die soziale und die symbolische. Diesen werden gegenübergestellt drei inhärente werte der technik, nämlich effizienz, ökonomie (*economy*) und eleganz. Dann ordnet er diese werte den unterschiedlichen dimensionen zu: die effizienz, welche definiert ist als minimierung des ressourcenverbrauchs, gehört zur wissenschaftlichen dimension der technik. Ökonomie dagegen ist ein soziales ziel, bei dem der eingesetzte aufwand in relation gesetzt wird zum erzielten nutzen. Man beachte, dass sich dies von dem ziel der effizienz unterscheidet: selbst wenn der ingenieur beweist, dass ein bestimmtes ziel wirklich mit minimalem ressourcenverbrauch erreicht wurde, kann trotzdem die gesellschaftliche analyse zu dem schluss kommen, dass das erzielte resultat den eingesetzten aufwand nicht wert ist. Den wert „eleganz" ordnet MacLennan der symbolischen dimension der technik zu, indem er sagt: „Eleganz symbolisiert einen guten entwurf". Er diskutiert dies am beispiel der Tacoma Narrows brücke, die auch in der softwaretechnik-literatur sehr häufig zitiert wird. Diese brücke war eine der ersten, welche mit hilfe ausgedehnter computerberechnungen entworfen wurde. In einem unwetter 1940 brach die brücke jedoch zusammen; es existieren spektakuläre filmaufnahmen von diesem unglück. Nachfolgende analysen zeigten, dass sämtliche computerberechnungen der statischen sicherheit der brücke vollkommen korrekt waren; es war lediglich aufgrund des mangelnden fortschritts der damaligen technik darauf verzichtet worden, auch ein dynamisches modell der brücke durchzurechnen. Durch die von dem unwetter induzierten schwingungen wurde der zusammenbruch der brücke herbeigeführt. MacLennan (wie wahrscheinlich auch Billington) bemerkt, dass vor dem einsatz der computerprogramme der resultierende entwurf der brücke vermutlich aus eleganzerwägungen verworfen worden wäre. Daraus leiten sie den ratschlag ab, man möge seine aufmerksamkeit auf entwürfe beschränken, die gut *aussehen*, weil sie vermutlich dann auch gut *sind*. Auf der seite der bauingenieure lässt sich dies dadurch begründen, dass die mathematische analyse der probleme praktisch niemals vollständig sein kann. Die relevanten gleichungen werden oft vereinfacht, um überhaupt numerisch handhabbar zu werden und wir haben es sehr häufig mit sog. unterdeterminierten problemen zu tun. Eleganz kann der schlüssel zu einem anderen technikverständnis sein: Die belebte natur ist unter dem gesichtspunkt der evolution daraufhin konstruiert, jeweils optimale lösungen für bestimmte probleme durchzusetzen. So wie heute aerodynamiker und roboterkonstrukteure von lösungen aus dem tier- und pflanzenreich lernen, welche sich im laufe der evolution durchgesetzt haben, so können vermutlich allgemein ingenieure von dem im menschlichen geist (vermutlich vor dem hintergrund der optimalen auswahl im reproduktionsprozess) eingebauten sinn für eleganz profitieren. Die technischen lösungen, welche von einer mehrzahl der

menschen als elegant empfunden werden, sind vermutlich auch die technisch erfolgreichen und damit richtigen lösungen.

Ein ganz anderer vergleich wird von Valk (1997) gezogen, der ähnlichkeiten zwischen informatikern und *juristen* erkennt: wie der jurist arbeitet auch der informatiker in einem wohldefinierten formalen system von regeln, die seinem auftraggeber nicht ohne weiteres verständlich sind. Formales vorgehen und soziale wirklichkeit stehen zunächst beziehungslos nebeneinander oder gar im widerspruch zueinander. Die beratungs- und vermittlungskompetenz beider berufsgruppen ist daher in besonderem mass gefordert. Programmierung ist wie gesetzgebung ein prozess, bei dem ein gewünschtes verhalten auf eine art und weise spezifiziert wird, die nicht unmittelbar einsichtig erkennen lässt, ob der gewünschte effekt erzielt wird.

C.3 Geschichte der versionshaltung

Bereits 1975 entstand das werkzeug *SCCS (Source Code Control System)* von Marc J. Rochkind, das heute nicht mehr weiterentwickelt und auch nicht verwendet wird, obwohl es in den Unix-distributionen immer noch enthalten ist. Ein bleibendes detail von SCCS sind aber die *identification keywords*: In den verwalteten dateien schaute SCCS beim einchecken auf zeichenketten, die mit dollarzeichen beginnen und enden, und falls es dazwischen die bezeichnung einer seiner variablen findet (z.B. `Id:` für die identifikation der datei, `Author:` für den (letzten) autor, `Revision:` für die versionsnummer, `Log:` für die komplette änderungsgeschichte u. a. m.), dann ersetzte es diese zeichenkette durch die jeweils nach dem eincheck-vorgang relevante information. (Beachte aber, dass sich die datei dadurch erneut änderte!)

Abb. 67 zeigt den anfang einer übersicht über die versionsgeschichte einer datei des Informatik-I-buchs von Klaeren und Sperber (2002).

Es war lange zeit tradition, in jeder quelle und auch in jedem objektmodul bestimmte zeichenketten einzubetten, die vom versionshaltungssystem verwaltet und jeweils beim einbuchen angepasst wurden[50]. Ein programm `ident` konnte dann auskunft über die in einer datei enthaltenen komponenten geben; siehe abb. 68.

Beim SCCS wurde im repositorium jeweils die erste version der datei abgelegt zusammen mit einer sammlung von skripten, um hieraus jede beliebige nachfolgende version erzeugen zu können; dazu kam natürlich die „verwaltungsinformation“ mit den *change comments* etc. Bei häufig geänderten dateien stellte sich dieses verfahren als unpraktisch heraus, da zum auschecken einer datei immer längere zeit durch die editiervorgänge benötigt wurde. Das jün-

[50]Eine klassische wendung in C-programmen war `static char sccsid[] =`

```
RCS file: /afs/informatik.uni-tuebingen.de/home/sperber/cvs/problemprogramm/i1vorw.tex,v
Working file: i1vorw.tex
head: 1.13
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 13;    selected revisions: 13
description:
----------------------------
revision 1.13
date: 2001/11/28 13:45:08;  author: sperber;  state: Exp;  lines: +11 -5
Release.
----------------------------
revision 1.12
date: 2001/08/31 06:59:15;  author: klaeren;  state: Exp;  lines: +2 -2
Small stylistic improvement
----------------------------
revision 1.11
date: 2001/08/29 17:56:21;  author: klaeren;  state: Exp;  lines: +3 -2
Added tnx to Volker :-)
----------------------------
revision 1.10
date: 2001/08/28 18:35:45;  author: klaeren;  state: Exp;  lines: +6 -6
One ambiguous sentence corrected.  Some typos corrected.
----------------------------
revision 1.9
date: 2001/08/28 15:37:20;  author: sperber;  state: Exp;  lines: +42 -1
Danksagungen.
----------------------------
revision 1.8
date: 2001/08/26 18:08:13;  author: sperber;  state: Exp;  lines: +55 -62
Gründlich überarbeitet.
... ...
```

Abbildung 67: Versionsgeschichte einer datei

```
[lepuy] 21) ident ~/bin/rexx
/afs/informatik.uni-tuebingen.de/home/klaeren/bin/rexx:
     $Id: memory.c,v 1.14 1993/05/10 06:14:04 anders Exp anders $
     $Header: /auto/home/flipper/anders/flipper/prosj/rexx/src/RCS/signals.c,v 1.4
        1993/05/10 06:04:52 anders Exp anders $
     $Id: funcs.c,v 1.14 1993/05/10 06:10:22 anders Exp anders $
     $Header: /auto/home/flipper/anders/flipper/prosj/rexx/src/RCS/library.c,v 1.2
        1993/02/09 18:15:09 anders Exp anders $
     $Id: misc.c,v 1.8 1993/05/10 06:20:23 anders Exp anders $
     $Id: parsing.c,v 1.13 1993/05/10 06:11:00 anders Exp anders $
     $Id: tracing.c,v 1.9 1993/05/10 05:49:42 anders Exp anders $
     $Id: files.c,v 1.13 1993/05/10 06:19:06 anders Exp anders $
     $Id: interp.c,v 1.7 1993/05/07 20:23:38 anders Exp anders $
     $Id: lexsrc.l,v 1.13 1993/05/10 05:51:00 anders Exp anders $
     $Id: yaccsrc.y,v 1.15 1993/05/07 21:31:04 anders Exp anders $
     $Id: variable.c,v 1.14 1993/05/10 06:16:49 anders Exp anders $
     $Id: strmath.c,v 1.7 1993/05/10 06:06:02 anders Exp anders $
     $Id: expr.c,v 1.2 1993/05/10 05:52:08 anders Exp anders $
     $Id: interpret.c,v 1.16 1993/05/10 06:17:35 anders Exp anders $
     $Id: envir.c,v 1.2 1993/05/10 05:52:45 anders Exp anders $
     $Id: shell.c,v 1.14 1993/05/10 06:11:51 anders Exp anders $
     $Id: stack.c,v 1.13 1993/05/10 06:06:43 anders Exp anders $
     $Id: rexx.c,v 1.13 1993/05/10 06:09:38 anders Exp anders $
     $Id: macros.c,v 1.2 1993/05/10 06:04:06 anders Exp anders $
     $Id: builtin.c,v 1.14 1993/05/10 06:12:40 anders Exp anders $
     $Id: extlib.c,v 1.1 1993/05/07 21:36:04 anders Exp anders $
     $Header: /auto/home/flipper/anders/flipper/prosj/rexx/src/RCS/convert.c,v 1.7
         1993/05/07 21:23:08 anders Exp anders $
     $Id: unixfuncs.c,v 1.11 1993/05/10 06:08:07 anders Exp anders $
     $Id: error.c,v 1.9 1993/05/10 05:55:53 anders Exp anders $
```

Abbildung 68: Ident strings

gere *RCS (Revision control system)* ging deshalb umgekehrt vor: Fussend auf der beobachtung, dass in der regel die aktuelle version die interessanteste ist, wird diese im repositorium abgelegt zusammen mit skripten, die hieraus auf wunsch jede ältere version rekonstruieren können. SCCS und RCS gehen beide von der vorstellung aus, dass programmierer eine komponente aus dem repositorium entweder nur zum anschauen ausbuchen (dann können sie später keine neue version davon einbuchen) oder aber zum ändern ausbuchen (dann kann kein anderer programmierer die gleiche komponente zum ändern ausbuchen). Es sind dann mit jeder datei *locks* verbunden, die erst beim nachfolgenden checkin freigegeben werden.

Eine weitere Verbesserung war *CVS (Concurrent Versions System)*, das sich sich von dieser einschränkung freimachte und die parallelarbeit mehrerer programmierer an einem modul erlaubte. Es gab daher nur eine einzige methode des ausbuchens von komponenten. Wurde beim nächsten einbuchen festgestellt, dass sich die version im repositorium von derjenigen unterschied, die damals ausgebucht wurde, so versuchte das werkzeug von selbst, die versionen zu mischen. Wenn die parallelen änderungen in unterschiedlichen regionen der datei vorgenommen wurden, ging das auch gut; anderenfalls wurden konflikte gemeldet, die der (zweite) programmierer selbst auflösen musste, bevor er die komponente wieder einbuchen konnte. Es gab übrigens trotzdem die möglichkeit, dass ein autor eine datei ausdrücklich „verriegelte“ und so gegen änderungen durch andere schützte.

CVS war auch auf eine häufig angetroffene situation eingestellt: Ein softwareanbieter verteilt programme in quellform; diese werden lokal weiterentwickelt bzw. modifiziert. Nun sollen für den fall eines neuen releases durch den softwareanbieter die seit dem letzten release lokal vorgenommenen änderungen auf den neuen quellen wiederum nachgezogen werden. Es leuchtet ein, dass auch dies wie das mischen von parallel erfolgten änderungen eine anspruchsvolle aufgabe ist.

CVS war wie seine vorgängersysteme darauf abgestimmt, *einzelne dateien* zu verwalten, aber nicht *ganze projekte*. Natürlich ist ein ganzes projekt immer eine sammlung einzelner dateien, aber einige aufgaben in der verwaltung von projekten sind nur schwierig zu bewältigen, wenn man sich auf einzelne dateien konzentriert:

- Schwierigkeiten macht CVS beim umbenennen von dateien und beim verschieben von dateien aus einem verzeichnis in ein anderes. Für CVS lässt sich beides nicht anders beschreiben als dass eine datei gelöscht und eine andere neu der versionshaltung hinzugefügt wird; beides hat aber negative auswirkungen auf die darstellung der projektgeschichte[51].

[51] Ein geschickter administrator könnte aber auch unter umgehung des CVS im repositorium

- Ein weiteres problem beim CVS sind die *non-atomic commits*: Angenommen, ein programmierer hat eine gewisse menge von dateien geändert und will sie wieder ins repositorium einchecken. Das geht für einige dateien gut, aber plötzlich gibt es bei einer datei konflikte, die nicht so leicht aufzulösen sind, oder die netzwerkverbindung wird unterbrochen. Dann ist das repositorium in einem inkonsistenten zustand; es gibt für den verursacher keine einfache möglichkeit, seine bereits eingecheckten module wieder zurückzusetzen.

- Das nächste problem ist, dass CVS grundsätzlich innerhalb eines einzigen dateisystems arbeitet; alle mitarbeiter müssen schreibberechtigung für das verzeichnis des repositoriums haben, also insbesondere auch ein *login account* für das system. Häufig führt dies dazu, das weitere techniken (z. B. VPN-tunnel) benötigt werden.

- Beim CVS hat jede datei ihre eigene versionsnummer (s. abb. 68). Ein bestimmter softwarestand ist deswegen durch eine sammlung möglicherweise ganz unterschiedlicher versionsnummern charakterisiert und lässt sich damit nicht ohne weiteres rekonstruieren. CVS bietet dafür das konzept der *tags* an; damit kann ein bestimmter zustand des repositoriums markiert werden und somit später auch wieder aufgefunden werden. Wird es allerdings vergessen, im richtigen moment ein solches *tag* zu vergeben, so wird die rekonstruktion wieder schwieriger.

---

direkt dateien umbenennen oder verschieben, allerdings ein recht gefährlicher prozess

D Philosophische aspekte der softwaretechnik

Der titel dieses kapitels mag stark überzogen sein, „echte" philosophen werden sich sicher darüber aufregen. Was ich hier anbieten möchte, sind im wesentlichen lesehinweise auf material, das nicht direkt zum thema gehört, aber interessante ausblicke auf den geistigen hintergrund der softwaretechnik im speziellen oder der ingenieurwissenschaften im allgemeinen bietet.

Bob Colwell (2002), ein ehemaliger Intel-chefarchitekt, setzt sich mit „Murphy's Law" auseinander, nach dem alles, was schiefgehen *kann*, auch schiefgehen *wird*. Colwell, der selbst zugibt, für gewisse Pentium-entwurfsfehler verantwortlich zu sein, rückt hier das häufig glorifizierte bild des ingenieurs zurecht, der angeblich alles immer richtig macht und betont die rolle von katastrophen und beinahe-katastrophen in der weiterentwicklung der technik. Dabei gibt er eine fülle von lesehinweisen, von denen ich einige hier (mit seinen empfehlungen) weitergeben möchte. Petroski (1985) betont, daß die lernkurve in den ingenieurwissenschaften typischerweise immer direkt nach irgendwelchen unglücken am steilsten ist. Kranz (2000) schildert die hintergründe der raumfahrtprogramme Mercury, Gemini und Apollo und dabei insbesondere die entscheidungen, häufiger auch einen test mit völlig unbekanntem ausgang und relativ hohem risiko einzugehen. Der soziologe Perrow (1999) weist darauf hin, wie das versagen komplexer systeme ziemlich direkt vom *kopplungsgrad* zwischen seinen komponenten abhängt. Chiles (2001) weist schließlich darauf hin, daß Murphy's gesetz in der o.a. form falsch ist. Schlimm wird es jedoch, wenn beinahe-katastrophen nicht ernstgenommen werden; dann kann das weiterverfolgen der fehlerhaften technik tatsächlich sehenden auges in die katastrophe führen, wie an der explosion der raumfähre Challenger vorgeführt wird.

Interessante gedanken hat auch Sodan (1998) anzubieten, der vom Yin und Yang der softwaretechnik, linkshirn und rechtshirn etc. schreibt.

D.1 Die ethik des softwareingenieurs

Es ist seit einiger zeit üblich, sich mit ethischen fragestellungen zu beschäftigen. Gerade im zusammenhang mit gentechnologie, cloning, intensivmedizin etc. werden in großer zahl professuren für ethik in der biologie, ethik in der medizin, ethik in den wissenschaften gegründet. Wir können in der informatik nicht qualifiziert über ethik sprechen. Es gibt jedoch zahlreiche angebote, gerade auch in Tübingen in anderen fakultäten, welches man bei entsprechendem interesse nutzen sollte. An dieser stelle möchte ich nur einige wenige grundtatsachen zur ethik (aus dem Schülerduden Philosophie) referieren, sowie standescodices der IEEE (IEEE Computer Society TCSE 1996), ACM (*Code*

of Ethics and Professional Conduct, Gotterbarn u. a. (1999b), neuerdings vereinigt zu einem gemeinsamen „Code of Ethics“, Gotterbarn u. a. (1999a)) und der GI (Ethische Leitlinien, Coy u. a. (1993); Rödiger und Wilhelm (1996)) kurz ansprechen.

Lesenswert auch der artikel von Berenbach und Broy (2009).

— **Baustelle!** —

E Zitate zur Softwaretechnik

Diese zitate sind chronologisch geordnet. Die deutschen übersetzungen sind von mir selbst; zur sicherheit ist in der regel auch der originaltext in kursivschrift wiedergegeben, da jede übersetzung immer auch interpretation ist. In einzelfällen, wo die deutsche übersetzung im text steht, ist hier nur das originalzitat aufgeführt.

Informatikstudierende sind nicht immer als fleißige leser bekannt, aber ich möchte trotzdem die empfehlung aussprechen, diese zitate, die hier völlig aus dem zusammenhang gerissen sind, möglichst an der originalstelle nachzulesen.

1. *Architecti est scientia pluribus disciplinis et variis eruditionibus ornata. [...] Itaque eum etiam ingeniosum oportet esse et ad disciplinam docilem. Neque enim ingenium sine disciplina aut disciplina sine ingenio perfectum artificem potest efficere. Et ut litteratus sit, peritus graphidos, eruditus geometria, historias complures noverit, philosophos diligenter audierit, musicam scierit, medicinae non sit ignarus responsa iurisconsultorum noverit, astrologiam caelique rationes cognitas habeat.*

   L. Vitruvius Pollio, De architectura liber primus

2. Vor eine frage gestellt, die wir vollständig verstanden haben, müssen wir sie von jeder überflüssigen darstellung abstrahieren, sie auf ihre einfachstmögliche form reduzieren und sie in möglichst kleine teile zerlegen, die wir dann aufzählen. (René Descartes, 1637)

   Placés devant une question parfaitement comprise, nous devons l'abstraire de toute représentation superflue, la réduire à sa forme la plus simple, et la diviser en parties aussi petites que possibles dont on fera l'énumération.

3. Als mein Großvater seine erste Maschine gebaut hatte, stand er andächtig vor seinem Werk. Und siehe, es war alles kräftig und stark. Und dann fragte er sich, was wohl kaputtgehen könne, wenn etwas kaputtgeht. Er hatte zwei Jahre an dieser Maschine konstruiert und sie eigenhändig gebaut, und nun fand er nichts, was am schwächsten war. Weil er aber kein Verkäufer, sondern Ingenieur war, wußte er, daß doch einmal etwas kaputtgehen wird. Zum Beispiel, wenn ein Schraubenschlüssel in die Zahnräder fallen würde. Dann wäre es ihm lieber gewesen, es würde dasjenige kaputtgehen, das am billigsten ist. Da trieb er aus dem großen Schwungrad die Keilbefestigungen heraus und ersetzte sie durch einen Kupferstift. Und nun wußte er, daß es in aller Zukunft nur der Kupferstift sein kann, der kaputtgeht, wenn etwas in das Räderwerk gerät. (Alexander Spoerl, Gentlemen können es selbst, 1969)

4. Es ist eine sehr beschämende erfahrung, einen fehler für viele millionen Dollar zu machen, aber man erinnert sich stark daran. Ich erinnere mich lebhaft an den abend, als wir über die organisation der niederschrift der externen spezifikationen für OS/360 entschieden. Der leiter der architektur, der leiter der steuerprogramm-implementierung und ich verabschiedeten den plan, die termine und die verteilung der verantwortung.

   Der architekturleiter hatte 10 gute leute. Er versicherte, sie könnten die spezifikation schreiben und es richtig machen. Es würde zehn monate dauern, drei mehr, als im terminplan vorgesehen.

Der steuerprogrammleiter hatte 150 gute leute. Er versicherte, sie könnten die spezifikationen mit der koordination der architekturabteilung herstellen; sie würden von guter qualität sein und praktikabel, und er könnte es rechtzeitig erledigen. Außerdem würden seine 150 leute, wenn die architekturabteilung den auftrag bekäme, zehn monate dasitzen und däumchen drehen.

Darauf antwortete der architekturleiter, daß das resultat, wenn ich der steuerprogrammabteilung die verantwortung überließe, eben *nicht* rechtzeitig fertig würde, sondern auch drei monate zu spät, und mit viel schlechterer qualität. Ich tat es so, und genau das passierte. Er hatte in beiden beziehungen recht. Außerdem machte das fehlen eines einheitlichen konzepts das system teurer zu erstellen und zu ändern, und ich würde schätzen, daß es die fehlerbeseitigung ein ganzes jahr verlängert hat. (Fred Brooks (1975))

It is a very humbling experience to make a multimillion-dollar mistake, but it is also very memorable. I vividly recall the night we decided how to organize the actual writing of external specifications for OS/360. The manager of architecture, the manager of control program implementation, and I were threshing out the plan, schedule, and division of responsibilities.

The architecture manager had 10 good men. He asserted that they could write the specifications and do it right. It would take ten months, three more than the schedule allowed.

The control program manager had 150 good men. He asserted that they could prepare the specifications, with the architecture team coordinating; it would be well-done and practical, and he could do it on schedule. Furthermore, if the architecture team did it, his 150 men would sit twiddling their thumbs for ten months.

To this the architecture manager responded that if I gave the control program team the responsibility, the result would not *in fact be on time, but would also be three months late, and of much lower quality. I did, and it was. He was right on both counts. Moreover, the lack of conceptual integrity made the system far more costly to build and change, and I would estimate that it added a year to debugging time.* (Fred Brooks (1975))

5. *It is simply not possible to fix today what the environment should be like in the future, and then to steer the piecemeal process of development toward that fixed, imaginary world. (C. Alexander, 1975)*

6. *Each pattern describes a problem which occurs over and over again in our environment, and describes the core of the solution to that problem, in such a way that you can use the solution a million times over, without ever doing the same thing twice. (Alexander u. a. 1977)*

7. *Get into a rut early: Do the same processes the same way. Accumulate idioms. Standardize. The only difference (!) between Shakespeare and you was the size of his idiom list - not the size of his vocabulary. (Perlis 1982)*

8. *In early 1982, the Lisa software team was trying to buckle down for the big push to ship the software within the next six months. Some of the managers decided that it would be a good idea to track the progress of each individual engineer in terms of the amount of code that they wrote from week to week. They devised a form that each engineer was required to submit every Friday, which included a field for the number of lines of code that were written that week.*

   Bill Atkinson, the author of Quickdraw and the main user interface designer, who was by far the most important Lisa implementor, thought that lines of code was a silly measure of software productivity. He thought his goal was to write as small and fast a program as possible, and that the lines of code metric only encouraged writing sloppy, bloated, broken code.

   He recently was working on optimizing Quickdraw's region calculation machinery, and had completely rewritten the region engine using a simpler, more general algorithm which, after some

tweaking, made region operations almost six times faster. As a by-product, the rewrite also saved around 2,000 lines of code.

He was just putting the finishing touches on the optimization when it was time to fill out the management form for the first time. When he got to the lines of code part, he thought about it for a second, and then wrote in the number: -2000.

I'm not sure how the managers reacted to that, but I do know that after a couple more weeks, they stopped asking Bill to fill out the form, and he gladly complied. (Hertzfeld 1982)

9. *Anything can be made measurable in a way that is superior to not measuring it at all.* (Gilb 1988)

10. Das Verwerfen des defensiven Programmierens bedeutet, daß der Kunde und der Anbieter nicht beide für eine Konsistenzbedingung verantwortlich gemacht werden. Entweder ist die Bedingung Teil der Vorbedingung und muß vom Kunden garantiert werden oder sie ist in der Vorbedingung nicht erwähnt und muß vom Anbieter behandelt werden. Welche dieser beiden Lösungen sollte gewählt werden? Da gibt es keine absolute Regel; verschiedene Stile des Schreibens von Funktionen sind möglich, von „fordernden", wo die Vorbedingungen stark sind (und damit die Verantwortung auf die Kunden abwälzen) bis zu „toleranten", wo die Vorbedingungen schwach sind (und damit die Last der Funktion erhöhen). Die Auswahl zwischen den beiden ist eine Sache der persönlichen Präferenzen; das Kriterium ist wiederum, die Einfachheit der Architektur insgesamt zu maximieren. Die Erfahrung mit Eiffel, speziell mit dem Entwurf der Bibliotheken, legt nahe, daß der systematische Gebrauch des fordernden Stils ziemlich erfolgreich sein kann. In diesem Ansatz konzentriert sich jede Funktion darauf, einen wohldefinierten Auftrag möglichst gut auszuführen anstatt zu versuchen, jeden erdenklichen Fall zu behandeln. (Meyer 1992)

11. Im Jahr 1988 schloß ein Konsortium, bestehend aus Hilton Hotels Corporation, Marriott Corporation und Budget Rent-A-Car Corporation, einen Vertrag über ein Großprojekt mit AMR Information Services, einer Tochterfirma der American Airlines Corporation. Diese Unternehmensberatung sollte ein neues Informationssystem mit dem Namen CONFIRM als ein hochmodernes umfassendes Reiseindustrie-Reservierungsprogramm in Kombination von Fluglinien-, Mietwagen- und Hotelinformation entwickeln. Ein neues Unternehmen, Intrico, wurde speziell für den Betrieb des neuen Systems gegründet. Das Konsortium hatte große Pläne, den Dienst auch an andere Gesellschaften zu vermarkten, aber als Hilton das System testete, traten größere Probleme auf. Wegen der Fehlfunktionen kündigte Intrico eine 18-monatige Verzögerung an. Die Probleme konnten aber nicht gelöst werden, und dreieinhalb Jahre nach Projektbeginn und nachdem eine Gesamtsumme von 125 Millionen Dollar investiert worden waren, wurde das Projekt abgebrochen. Oz (1994)

12. Obwohl unsere Industrie reich ist an Beispielen von Software-Systemen, die schlecht funktionieren, voll von Fehlern sind oder anderweitig darin versagen, die Wünsche der Benutzer zu erfüllen, gibt es Gegenbeispiele. Große Softwaresysteme *kann* man mit sehr hoher Qualität herstellen, aber sie sind sehr teuer — in der Gegend von 1.000 Dollar pro Programmzeile. Ein Beispiel ist die Bord-Software von IBM für das Space Shuttle: drei Millionen Programmzeilen mit weniger als einem Fehler pro 10.000 Zeilen. (Davis 1994)

13. *...computers do not introduce new forms of error, but they increase the scope for introducing conventional errors by increasing the complexity of the processes that can be controlled. In addition, the software controlling the process may itself be unnecessarily complex and tightly coupled. Adding even more complexity in an attempt to make the software "safer" may cause more accidents than it prevents. Proposals for safer software design need to be evaluated as to whether any added*

complexity is such that more errors will be introduced than eliminated and whether a simple way exists to achieve the same goal. (Leveson 1995)

14. *The big question mark in the future is formal methods. It is difficult here, for someone like me who became a computer scientist by working on abstract data types and the original Z, to avoid mistaking wishful thinking for technology assessment. It is clear to all the best minds in the field that a more mathematical approach is needed for software to progress much. But this is not acepted by the profession at large. I can see two possible scenarios. The best one is that software science education will involve higher doses of formalism and that this will translate over time into a more formal approach in the industry at large. The other scenario is that formal techniques will be used only in high-risk development projects controlled by governmental regulatory agencies and will continue to exert some influence on the better programming languages. I must say that, wishful thinking aside, the last scenario is more likely—barring the unhappy prospect of a widely publicized, software-induced catastrophe.* (Meyer 1995)

15. Als beispiel dafür, was manager verstehen müssen, betrachten wir ein in der industrie übliches maß für die produktivität, nämlich das verhältnis von produziertem code zum produktionsaufwand. Ein prozeß mit berücksichtigung von wiederverwendung würde vermutlich einige zeit darauf verwenden, softwareelemente, die bereits funktionieren, zu verbessern, in der absicht, ihr potential für die wiederverwendung in zukünftigen projekten zu verbessern. Dieser verallgemeinerungsschritt ist ein wichtiger schritt im objektorientierten lebenslauf. Solche anstrengungen werden oft code entfernen und damit den zähler (= code) des produktivitätsmaßes verringern und den nenner (= aufwand) vergrößern. Die manager müssen gewarnt werden, daß die alten maße nicht die ganze geschichte erzählen und daß der zusätzliche aufwand in der tat das softwareguthaben der gesellschaft vergrößert. (B. Meyer (1996))

    As an example of what managers must understand, consider a common industry measure of productivity: the ratio of produced code to production effort. A reuse-conscious process may spend some time improving software elements that already work well to increase their potential for reuse in future projects. This generalization task *is an important step in the OO life cycle. Such efforts will often remove code, decreasing the productivity ratio's numerator (code) and thus increasing the denominator (effort)! Managers must be warned that old measures do not tell the whole story and that the extra effort actually improves the software assets of the company.*

16. *A scientific theory is formalised as a mathematical model of reality, from which can be deduced or calculated the observable properties and behaviour of a well-defined class of processes in the physical world. It is the task of theoretical scientists to develop a wide range of plausible but competing theories; experimental scientists will then refute or confirm the theories by observation and experiment. The engineer then applies a confirmed theory in the reverse direction: the starting point is a specification of the observable properties and behaviour of some system that does not yet exist in the physical world; and the goal is to design and implement a product which can be predicted by the theory to exhibit the specified properties. Mathematical methods of calculation and proof are used throughout the design task.* (Hoare 1993)

17. *Frequently, capacity and performance goals get shelved during development. After the system is built, we push it off a cliff and see if it flies. It would be better to keep capacity goals in mind during design/development and to get performance feedback all along.* (Mackey 1996)

18. *Some people assume that the curriculum should start with object-oriented analysis. This is a grave mistake. A beginner cannot understand OO analysis [...] To master OO analysis, you must first master fundamental concepts, like class, contracts, information hiding, inheritance, polymorphism, dynamic binding, and the like, at the level of implementation, where they are*

immediately. You must also have used these concepts to build a few OO systems, first small and then larger, all the way to completion. Only after such a hands-on encounter with the operational use of the method will you be equipped to understand the concepts of OO analysis and their role in the seamless project of object-oriented software construction. Initial training, then, should focus on implementation and design. (Meyer 1996)

19. Nehmen wir an, Sie haben ein System in dem jedes Objekt eine eindeutige ID haben soll, die in einer Attributvariablen namens ID gespeichert ist. Ein Entwurfsansatz würde es sein, ganze Zahlen für die IDs zu verwenden und die höchste vergebene ID in einer globalen Variablen `maxID` zu speichern. An jeder Stelle, wo ein neues Objekt alloziert wird, [. . .] könnte man einfach die Anweisung `ID = ++maxID` verwenden. [. . .] Was könnte daran falsch sein? Eine Menge! Was, wenn Sie einen bestimmten Bereich von IDs für spezielle Zwecke reservieren möchten? Was, wenn Sie IDs von Objekten wiederverwenden möchten, die gelöscht wurden? Was, wenn Sie eine Zusicherung einfügen möchten, die einen Alarm auslöst, wenn Sie mehr IDs allozieren wollen als die maximal vorgesehene Anzahl? Wenn Sie IDs alloziert haben, indem Sie Anweisungen `ID = ++MaxID` durch das ganze Programm verstreut haben, dann müßten Sie den Code von jeder dieser Anweisungen ändern. Die Art, wie neue IDs erzeugt werden, ist eine Entwurfsentscheidung, die Sie verstecken sollten. Wenn Sie die Phrase `++MaxID` über das Programm hinweg verwenden, dann decken Sie auf, daß eine neue ID durch Erhöhen von `maxID` erzeugt wird. Wenn Sie statt dessen die Anweisung `ID = NewID()` im Programm verwenden, dann verbergen Sie die Information, wie neue IDs erzeugt werden. Innerhalb der Funktion `NewID()` könnten Sie immer noch eine einzige Zeile — `return (++MaxID)` oder ähnlich — haben, aber wenn Sie später beschließen, bestimmte Bereiche von IDs für spezielle Zwecke zu reservieren oder alte IDs wiederzuverwenden, dann könnten Sie diese Änderungen innerhalb der `NewID`-Funktion machen, ohne Dutzende oder gar hunderte von Anweisungen anzufassen. [. . .] Jetzt nehmen Sie an, daß Sie feststellen, daß Sie den Typ ID von einer ganzen Zahl zu einer Zeichenkette verwandeln müssen. Wenn Sie Deklarationen der Form `int ID` durch das Programm verteilt haben, dann hilft Ihre Verwendung der `NewID`-Funktion nicht weiter. Sie müssen immer noch durch das Programm gehen und Dutzende oder Hunderte von Änderungen machen. In diesem Fall ist der Typ von ID die Entwurfsentscheidung, die Sie verstecken müssen. Sie könnten einfach Ihre IDs als vom Typ `IDTYPE` deklarieren, einem benutzerdefinierten Typ. (McConnell 1996)

20. The field has been defined by the Engineers Council for Professional Development, in the United States, as the creative application of „scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behaviour under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property.“ [. . .] The function of the scientist is to know, while that of the engineer is to do. The scientist adds to the store of verified, systematized knowledge of the physical world; the engineer brings this knowledge to bear on practical problems. [. . .] Unlike the scientist, the engineer is not free to select the problem that interests him; he must solve problems as they arise; his solution must satisfy conflicting requirements. Usually efficiency costs money; safety adds to complexity; improved performance increases weight. The engineering solution is the optimum solution, the end result that, taking many factors into account, is most desirable. It may be the most reliable within a given weight limit, the simplest that will satisfy certain safety requirements, or the most efficient for a given cost. In many engineering problems the social costs are significant. (Encylopædia Britannica, 1999)

21. **SMART: Programm stabilisiert Auto.** Renningen. Der Kleinwagen Smart wird mit dem Stabilisierungsprogramm Trust Plus nachgerüstet. Der Smart-Produzent Micro Compact Car (MCC/Renningen) teilte gestern mit, die Maßnahme sei das Ergebnis einer Überprüfung des Winterfahrverhaltens. Unter extremen Bedingungen waren einige Fahrzeuge umgekippt. Das neue Stabilisierungsprogramm nehme zusätzlich zur bereits vorhandenen Stabilisierungskontrolle auch während des Beschleunigens Einfluß auf Motor und Kupplung und regele die Geschwindigkeit der angetriebenen Räder. „Das Nachrüsten geschieht durch einfaches Nachladen der Software“, sagte ein Smart-Sprecher. Bei bisher ausgelieferten Fahrzeugen wird das System kostenlos nachgeladen. (Südwestpresse, 26.1.1999)

22. **Extreme Programming Core Practices** *The 12 XpXtudes of ExtremeProgramming grouped into four categories*

    - *Fine scale feedback*
      - *TestDrivenDevelopment via ProgrammerTests and CustomerTests (were UnitTests & AcceptanceTests)*
      - *PlanningGame*
      - *WholeTeam (was OnsiteCustomer)*
      - *PairProgramming*
    - *Continuous process rather than batch*
      - *ContinuousIntegration*
      - *DesignImprovement (was RefactorMercilessly)*
      - *SmallReleases*
    - *Shared understanding*
      - *SimpleDesign (DoSimpleThings, YouArentGonnaNeedIt, OnceAndOnlyOnce, SimplifyVigorously)*
      - *SystemMetaphor*
      - *CollectiveCodeOwnership*
      - *CodingStandard or CodingConventions*
    - *Programmer welfare*
      - *SustainablePace (original name: FortyHourWeek)*

    (`http://c2.com/cgi/wiki?ExtremeProgrammingCorePractices`, Jun. 2007)

23. **Developer Bill Of Responsibilities** *You have the responsibility to:*

    - *understand what is needed, and why it is your customer's priority.*
    - *produce quality work even when you are pressed to do otherwise*
    - *mentor others and impart whatever skills and wisdom you may have*
    - *make your estimates as accurate as possible, know when you are off schedule, why, and how to make the schedule to reflect reality as quickly as possible.*
    - *accept the consequences of your actions*

    (`http://c2.com/cgi/wiki?DeveloperBillOfResponsibilities`, Aug. 2006)

24. **Customer Bill of Rights** *You have the right to:*

(a) *Declare the business priority of every UserStory*
(b) *Have an overall plan, to know what can be accomplished, when, and at what cost.*
(c) *Get the most possible value out of every programming week.*
(d) *See progress in a running system, proven to work by passing repeatable tests that you specify.*
(e) *Advise developers of changes in the requirements*
(f) *Be informed of schedule changes, in time to choose how to reduce scope to restore the original date.*
(g) *Cancel project at any time and be left with a useful working system reflecting investment to date.*

(`http://c2.com/cgi/wiki?CustomeBillOfRights`, Mar. 2010)

25. **Developer Bill of Rights** *You have the right to:*

- *Know what is needed, with clear declarations of priority in the form of detailed requirements and specifications. In* ExtremeProgramming*, the customer can change the requirements and specifications by adding or removing stories, or change the priority of stories. The programmer gets to see the new stories, estimate them, and inform that customer of the impact on the schedule. If the schedule no longer meets the customer's desired ship date, then the developer informs the customer, and the customer can either accept the new date, or remove enough lower priority stories to make the date. It is not a disaster for the customer to make changes, it is an inevitability.*
- *Have clear and continuing communications with the client, both the end user and the "responsible authority."*
- *Produce quality work at all times and have support for doing a quality job, even if it takes a little longer and requires buying tools.*
- *Ask for and receive help from peers, superiors, and customers and have time and opportunities built into the schedule for communicating with other project members.*
- *Make, and update your own estimates including having an input into your long range schedule and goals.*
- *Have responsibility for your own day-to-day scheduling and goals.*
- *Have management/client support for continuing education including but not limited to: books, subscriptions, time and money to try new programming tools, meetings, training, etc.*
- SustainablePace/FortyHourWeek*s*
- *Use your own development tools where appropriate as long as the end result is compatible with your* CustomersExpectations*.*

(`http://c2.com/cgi/wiki?DeveloperBillOfRights`, Oct. 2008)

26. *One of the things that makes engineering to interesting—besides the fact that they'll pay you to do it—is the interplay between so many contravening influences: features, performance, technology, risk, schedule, design team capability, and economics. You can find a perfect balance among all of these, and still fail, because you designed something the buying public happens not to want.* Colwell (2004)

27. *The XP ideal, as Beck says, is mutual respect and collaboration among programmers. In contrast, at Microsoft and most other software companies, we often see marketing goals take priority over professional goals in a project, forcing companies to use overtime to compensate for unrealistic schedules or poor project management and teamwork. Another approach common in Windows and other Microsoft groups, rather than simply relying on overtime, is to delay products, sometimes for years.* Cusumano (2007a)

28. *I believe iterative development practices are useful for any software project where the upfront specifications are incomplete – which is nearly always the case.* Cusumano (2007a)

Literatur

Adams 1984 ADAMS, Edward N.: Optimizing Preventive Service of Software Products. In: *IBM J. Res. Develop.* 28 (1984), Nr. 1, S. 2–14

Alexander u. a. 1977 ALEXANDER, C. ; ISHIKAWA, S. ; SILVERSTEIN, M. ; JACOBSON, M. ; FIKSDAHL-KING, I. ; ANGEL, S.: *A Pattern Language.* New York : Oxford University Press, 1977

Armour 2005 ARMOUR, Philip G.: The business of software: The unconscious art of software testing. In: *Communications of the ACM* 48 (2005), Nr. 1, S. 15–18

Bauer 1991 BAUER, Friedrich L.: Informatik und Algebra. In: BROY, M. (Hrsg.): *Informatik und Mathematik*, Springer-Verlag, 1991, S. 28–40. – Vortrag beim Festkolloquium „20 Jahre Institut für Informatik“, Zürich 1988

Bauer 1993 BAUER, Friedrich L.: Software Engineering — wie es begann. In: *Informatik-Spektrum* 16 (1993), S. 259–260

Beck 1999 BECK, Kent: Embracing Change with Extreme Programming. In: *IEEE Computer* 32 (1999), Nr. 10, S. 70–77

Bentley 1986 BENTLEY, Jon: *Programming Pearls*. Addison-Wesley, 1986

Bentley 1988 BENTLEY, Jon: *More Programming Pearls*. Addison-Wesley, 1988

Berenbach und Broy 2009 BERENBACH, Brian ; BROY, Manfred: Professional and Ethical Dilemmas in Software Engineering. In: *IEEE Computer* 42 (2009), Nr. 1, S. 74–80

Berry und Kamsties 2005 BERRY, Daniel M. ; KAMSTIES, Erik: The Syntactically Dangerous *All* and Plural in Specifications. In: *IEEE Software* 22 (2005), Nr. 1, S. 55–57

Bessey u. a. 2010 BESSEY, Al ; BLOCK, Ken ; CHELF, Ben ; CHOU, Andy ; FULTON, Bryan ; HALLEM, Seth ; HENRI-GROS, Charles ; KAMSKY, Asya ; MCPEAK, Scott ; ENGLER, Dawson: A few billion lines of code later: using static analysis to find bugs in the real world. In: *Commun. ACM* 53 (2010), Nr. 2, S. 66–75. – ISSN 0001-0782

Billington 1985 BILLINGTON, David P.: *The Tower and the Bridge*. Princeton University Press, 1985

Boehm und Basili 2001 BOEHM, Barry ; BASILI, Victor R.: Software Defect Reduction Top 10 List. In: *IEEE Computer* 34 (2001), Nr. 1, S. 135–137

Boehm und In 1996 BOEHM, Barry ; IN, Hoh: Identifying Quality-Requirement Conflicts. In: *IEEE Software* 13 (1996), Nr. 2, S. 25–35

Booch 1998 BOOCH, Grady: *The Unified Modeling Language User Guide*. Addison-Wesley, 1998

du Bousquet u. a. 2004 BOUSQUET, L. du ; LEDRU, Y. ; MAURY, O. ; ORIAT, C. ; LANET, J.-L.: A Case Study in JML-based Software Validation. In: *Proc. of 19th IEEE Conf. on Automated Software Engineering, ASE'04*. Linz : IEEE CS Press, 2004, S. 294–297

Bowen 2005 BOWEN, Jonathan: *The Z Notation*. http://vl.zuser.org/. Sep 2005

Bowen und Hinchey 2006 BOWEN, Jonathan P. ; HINCHEY, Michael G.: Ten Commandmends of Formal Methods. . . Ten Years Later. In: *IEEE Computer* 33 (2006), Nr. 1, S. 40–48

British Standards Institute 2011 BRITISH STANDARDS INSTITUTE: *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. British Standards Institute, 2011. – BS ISO/IEC 25010:2011, http://http://janus.uclan.ac.uk/pagray/BS-ISO-IEC%2025010%202011%20quality%20requirements%20models.pdf

Brooks 1975 BROOKS, Frederick P.: *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975

Brooks 1995 BROOKS, Frederick P.: The Mythical Man-Month After 20 Years. In: *IEEE Software* 12 (1995), Nr. 5, S. 57–60

Broy und Pree 2003 BROY, Manfred ; PREE, Wolfgang: Ein Wegweiser für Forschung und Lehre im Software-Engineering eingebetteter Systeme. In: *Informatik-Spektrum* 26 (2003), Nr. 1, S. 3–7

Cardelli und Wegner 1985 CARDELLI, Luca ; WEGNER, Peter: On Understanding Types, Data Abstraction, and Polymorphism. In: *ACM Computing Surveys* 17 (1985), Dezember, S. 471–522

Carey 1996 CAREY, Dick: Is Software Quality Intrisic, Subjective, or Relational. In: *ACM Software Engineering Notes* 21 (1996), Nr. 1, S. 74–75

Chiles 2001 CHILES, James: *Inviting Disasters: Lessons From the Edge of Technology*. New York : Harper Business, 2001

Coad und Yourdon 1991 COAD, Peter ; YOURDON, Edward: *Object-Oriented Analysis*. Englewood Cliffs, New Jersey : Yourdon Press, 1991

Cobb und Mills 1990 COBB, Richard H. ; MILLS, Harlan D.: Engineering Software under Statistical Quality Control. In: *IEEE Software* 7 (1990), Nr. 11, S. 44–54

Colwell 2002 COLWELL, Bob: Near Misses: Murphy's Law Is Wrong. In: *IEEE Computer* 35 (2002), Nr. 4, S. 9–12

Colwell 2004 COLWELL, Bob: Design Fragility. In: *IEEE Computer* 37 (2004), Nr. 1, S. 13–16

Coy u. a. 1993 COY, Wolfgang u. a.: Ethische Leitlinien der Gesellschaft für Informatik. In: *Informatik-Spektrum* 16 (1993), Nr. 4, S. 239–240

Cusumano 2007a CUSUMANO, Michael A.: Extreme Programming Compared with Microsoft-Style Interative Development. In: *Communications of the ACM* 50 (2007), Nr. 10, S. 15–18

Cusumano 2007b CUSUMANO, Michael A.: What Road Ahead for Microsoft the Company? In: *Communications of the ACM* 50 (2007), Nr. 2, S. 15–18

Cusumano und Selby 1995 CUSUMANO, Michael A. ; SELBY, Richard W.: *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. New York : The Free Press, Simon & Schuster, 1995

Cusumano und Selby 1996 CUSUMANO, Michael A. ; SELBY, Richard W.: *Die Microsoft-Methode: Sieben Prinzipien, wie man ein Unternehmen an die Weltspitze bringt*. Freiburg : Rudolf Haufe Verlag, 1996

Cusumano und Selby 1997 CUSUMANO, Michael A. ; SELBY, Richard W.: How Microsoft Builds Software. In: *Communications of the ACM* 40 (1997), Nr. 6, S. 53–61

Davis 1994 DAVIS, Alan M.: Fifteen Principles of Software Engineering. In: *IEEE Software* 11 (1994), Nr. 6, S. 94–96,101

Davis u. a. 1988 DAVIS, Alan M. ; BERSOFF, E. ; COMER, E.: A Strategy for Comparing Alternative Software Development Life Cycles. In: *IEEE Transactions on Software Engineering* 14 (1988), Nr. 10, S. 1453–1460

Descartes 1637 DESCARTES, René: *Discours de la Méthode pour bien conduire sa raison et chercher la verité dans les sciences*. Paris, 1637

Drake 1996 DRAKE, Thomas: Measuring Software Quality: A Case Study. In: *IEEE Computer* 29 (1996), Nr. 11, S. 78–87

Dunsmore u. a. 2003 DUNSMORE, Alastair ; ROPER, Marc ; WOOD, Murray: Practical Code Inspection Techniques for Object-Oriented Systems: An Experimental Comparison. In: *IEEE Software* 20 (2003), Nr. 4, S. 21–29

Dworschak 2009 DWORSCHAK, Manfred: Windows aus der Asche. In: *Spiegel online* (2009). – `http://www.spiegel.de/spiegel/0,1518,634334-2,00.html`, 6. Juli 2009

Dörner 1989 DÖRNER, Dietrich: *Die Logik des Mißlingens — Strategisches Denken in komplexen Situationen*. Rowohlt, 1989

EB99 1999 *"engineering"*. Encyclopædia Britannica Online. 1999. – `http://members.eb.com/bol/topic?eu=108127&query=engineering[AccessedApril13,1999]`

Ehrich u. a. 1989 EHRICH ; GOGOLLA ; LIPECK: *Algebraische Spezifikation abstrakter Datentypen*. Teubner, 1989

Ernst u. a. 1997 ERNST, Dietmar ; HOUDEK, Frank ; SCHULTE, Wolfram ; SCHWINN, Thilo: Experimenteller Vergleich der statischen und dynamischen Softwareprüfung / Universität Ulm, Fakultät für Informatik. 1997 (97-13). – Ulmer Informatik-Berichte

Everett 1980 EVERETT, Robert E.: Whirlwind. In: (Metropolis u. a. 1980), S. 365–384

Fagan 1976 FAGAN, M. E.: Design and code inspections to reduce errors in program development. In: *IBM Systems Journal* 15 (1976), Nr. 3, S. 182–211

Fayad und Schmidt 1997 FAYAD, M.E. ; SCHMIDT, D.C.: Object-Oriented Application Frameworks. In: *Communications of the ACM* 40 (1997), Nr. 10, S. 32–38

Feldman 1979 FELDMAN, Stuart I.: Make—A Program for Maintaining Computer Programs. In: *Software—Practice and Experience* 9 (1979), Nr. 3, S. 255–265

Floyd 1994 FLOYD, Christiane: Software-Engineering — und dann? In: *Informatik-Spektrum* 17 (1994), Nr. 1, S. 29–37

Floyd u. a. 1989 FLOYD, Christiane ; REISIN, F.-M. ; SCHMIDT, G.: STEPS to Software Development with Users. In: GHEZZI, C. (Hrsg.) ; MCDERMID, J. A. (Hrsg.): *ESEC'89*, 1989 (Lecture Notes in Computer Science 287), S. 48–64

Fowler u. a. 1999 FOWLER, Martin ; BECK, Kent ; BRANT, John ; OPDYKE, William ; ROBERTS, Don: *Refactoring. Improving the design of existing code*. Addison-Wesley, 1999

Fowler und Scott 1998 FOWLER, Martin ; SCOTT, Kendall: *UML Distilled. Applying the Standard Object Modeling Language*. Addison-Wesley, 1998

Gabriel 1997 GABRIEL, Richard P.: *Patterns of Software*. Oxford University Press, 1997. – ISBN 0-19-510269-X

Gamma u. a. 1995 GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

Gilb 1988 GILB, Tom: *Principles of Software Engineering*. Boston : Addison-Wesley, 1988

Glass 2005 GLASS, Robert L.: IT Failure Rates — 70% or 10–15 In: *IEEE Software* 22 (2005), Nr. 3, S. 112–110

Gotterbarn u. a. 1999a GOTTERBARN, Don ; MILLER, Keith ; ROGERSON, Simon: Computer Society and ACM Approve Software Engineering Code of Ethics. In: *IEEE Computer* 32 (1999), Nr. 10, S. 84–88

Gotterbarn u. a. 1999b GOTTERBARN, Don ; MILLER, Keith ; ROGERSON, Simon: Software Engineering Code of Ethics is Approved. In: *Communications of the ACM* 42 (1999), Nr. 10, S. 102–107

Graham 2002 GRAHAM, Dorothy: Requirements and Testing: Seven Missing-Link Myths. In: *IEEE Software* 19 (2002), Nr. 5, S. 15–17

Graham u. a. 1982 GRAHAM, Susan L. ; KESSLER, Peter B. ; MCKUSICK, Marshall K.: gprof: A Call Graph Execution Profiler. In: *SIGPLAN '82 Symposium on Compiler Construction* Bd. 17(6), 1982, S. 120–126

Guttag u. a. 1993 GUTTAG, John V. ; HORNING, James J. ; GARLAND, S. J. ; JONES, K. D. ; MODET, A. ; WING, J. M.: *Larch: Languages and Tools for Formal Specification*. Springer, 1993

Henning 2007 HENNING, Michi: API Design Matters. In: *ACM Queue* 5 (2007), Nr. 4, S. 25–36. – Also in CACM 52(5), 2009, p. 46-56

Hertzfeld 1982 HERTZFELD, Andy: *-2000 Lines of Code*. `http://www.folklore.org/StoryView.py?project=Macintosh&story=Negative_2000_Lines_Of_Code.txt`. Feb. 1982

Hoare 1969 HOARE, C. A. R.: An axiomatic basis for computer programming. In: *Communications of the ACM* 12 (1969), October, Nr. 10, S. 576–580

Hoare 1972 HOARE, C. A. R.: Proof of Correctness of Data Representations. In: *Acta Informatica* 1 (1972), S. 271–281

Hoare 1993 HOARE, C. A. R.: Algebra and Models. In: *Proceedings SIGSOFT'93* Bd. 18(5) ACM (Veranst.), 1993, S. 1–8

Hoare 2009 HOARE, C. A. R.: Retrospective: An Axiomatic Basis for Computer Programming. In: *Communications of the ACM* 52 (2009), October, Nr. 10, S. 30–32

Holbæk-Hanssen u. a. 1977 HOLBÆK-HANSSEN, E. ; HANDLYKKEN, P. ; NYGAARD, K.: System Description and the Delta Project / Norske Regnecentralen. Oslo : Norske Regnecentralen, 1977 (4). – Forschungsbericht

Holzmann 2003 HOLZMANN, G.: *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003

Humphrey 2000 HUMPHREY, Watt S.: The Personal Software Process: Status and Trends. In: *IEEE Software* 17 (2000), Nr. 6, S. 71–75

Höhn und Höppner 2008 HÖHN, Reinhard ; HÖPPNER, Stephan: *Das V-Modell XT – Anwendungen, Werkzeuge, Standards*. Springer, 2008

IEEE 1990 IEEE: *Standard Glossary of Software Engineering Terminology*. IEEE Standard 610.12-1990, 1990

IEEE Computer Society TCSE 1996 IEEE COMPUTER SOCIETY TCSE: Rough Draft of a Code of Ethics for Software Engineers. In: *Software Engineering Technical Council Newsletter* 14 (1996), Nr. 4, S. 4,49

Jones 1996 JONES, Capers: Software Defect-Removal Efficiency. In: *IEEE Computer* 29 (1996), Nr. 4, S. 94–95

Jones 1997 JONES, Capers: *Software Quality - Analysis and Guidelines for Success*. International Thomson Computer Press, 1997. – ISBN 1-85032-867-6

Jones 1990 JONES, Cliff B.: *Systematic Software Development using VDM*. 2. Aufl. Prentice Hall, 1990

Jézéquel und Meyer 1997 JÉZÉQUEL, Jean-Marc ; MEYER, Bertrand: Design by Contract: The Lessons of Ariane. In: *IEEE Computer* 30 (1997), Nr. 1, S. 129–130

Kernighan und Pike 1999 KERNIGHAN, Brian ; PIKE, Rob: Finding Performance Improvements. In: *IEEE Software* 16 (1999), Nr. 2, S. 61–68

Klaeren 1983 KLAEREN, Herbert: *Algebraische Spezifikation — Eine Einführung*. Berlin-Heidelberg-New York : Springer Verlag, 1983

Klaeren 1990 KLAEREN, Herbert: *Vom Problem zum Programm — Eine Einführung in die Informatik*. Teubner Verlag, 1990. – 2. Aufl. 1991

Klaeren 1994 KLAEREN, Herbert: Probleme des Software Engineering. Die Programmiersprache — Werkzeug des Softwareentwicklers. In: *Informatik-Spektrum* 17 (1994), S. 21–28

Klaeren 2006 KLAEREN, Herbert: *Viren, Würmer und Trojaner: Streifzüge durch die Computerwelt*. Tübingen : Klöpfer & Meyer, 2006

Klaeren und Sperber 2002 KLAEREN, Herbert ; SPERBER, Michael: *Vom Problem zum Programm: Architektur und Bedeutung von Computerprogrammen*. 3. Aufl. Teubner, 2002

Klaeren und Sperber 2007 KLAEREN, Herbert ; SPERBER, Michael: *Die Macht der Abstraktion: Einführung in die Programmierung*. Teubner, 2007

Klein u. a. 2004 KLEIN, Torsten ; CONRAD, Mirko ; FEY, Ines ; GROCHTMANN, Matthias: Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. In: *Proc. Modellierung 2004, Marburg* Bd. P-45, URL `http://www.immos-projekt.de/site_immos/download/p3_KCF+04.pdf`, 2004, S. 31–41

Kleppe u. a. 2003 KLEPPE, Anneke ; WARMER, Jos ; BAST, Wim: *MDA explained : the model driven architecture. Practice and promise*. Boston u.a. : Addison-Wesley, 2003 (The Addison-Wesley object technology series)

Knuth 1971 KNUTH, Donald E.: Empirical Study of Fortran Programs. In: *Software—Practice and Experience* (1971), S. 105–133

Koch 1998 KOCH, Richard: *The 80/20 Principle. The Secret of Achieving More with Less*. London : Nicholas Brealy Publishing, 1998

Kranz 2000 KRANZ, Gene: *Failure ist Not an Option*. New York : Berkley Books, Penguin Putnam, 2000

Krasner und Pope 1988 KRASNER, G.E. ; POPE, S.T.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. In: *Journal of Object-Oriented Programming* 1 (1988), August/ September, Nr. 3, S. 26 – 49

Leavens 2007 LEAVENS, Gary: *JML Reference Manual*. Iowa State University: , Feb. 2007. – `http://www.cs.iastate.edu/~leavens/JML/jmlrefman/`

Leavens und Cheon 2006 LEAVENS, Gary T. ; CHEON, Yoonsik: *Design by Contract with JML*. ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf/. Sep 2006

Leveson 1995 LEVESON, Nancy G.: Safety as a System Property. In: *Communications of the ACM* 38 (1995), Nr. 11, S. 146

Licklider 1969 LICKLIDER, J. C.: Underestimates and Overexpectations. In: CHAYES, Abram (Hrsg.) ; WIESNER, Jerome (Hrsg.): *ABM: An Evaluation of the Decision to Deploy and Anti-Ballistic Missile*. Harper & Row, 1969

Linger 1994 LINGER, Richard C.: Cleanroom Process Model. In: *IEEE Software* 11 (1994), Nr. 2, S. 50–58. – (Best practice paper from ICSE'15, selected by IEEE Software Editorial Board)

Louridas 2006 LOURIDAS, Panagiotis: Static Code Analysis. In: *IEEE Software* 23 (2006), Nr. 4, S. 58–61

Mackey 1996 MACKEY, Karen: Why Bad Things Happen to Good Projects. In: *IEEE Software* 13 (1996), Nr. 3, S. 27–32

MacLennan 1997 MACLENNAN, Bruce J.: Who Cares About Elegance? The Role of Aesthetics in Programming Language Design. In: *SIGPLAN Notices* 32 (1997), Nr. 3, S. 33–37

McCabe 1976 MCCABE, Thomas J.: A Complexity Measure. In: *IEEE Trans Soft Eng* SE-2 (1976), Dec, Nr. 4, S. 308–320. – http://www.literateprogramming.com/mccabe.pdf

McConnell 1996 MCCONNELL, Steve: Missing in Action: Information Hiding. In: *IEEE Software* 13 (1996), Nr. 3, S. 128–127

McCracken und Jackson 1982 MCCRACKEN, Daniel D. ; JACKSON, Michael A.: Life Cycle Concept Considered Harmful. In: *Software Engineering Notes* 7 (1982), Nr. 2, S. 29–32

Metropolis u. a. 1980 METROPOLIS, Nicholas C. (Hrsg.) ; HOWLETT, Jack (Hrsg.) ; ROTA, Gian-Carlo (Hrsg.): *A History of Computing in the Twentieth Century*. Academic Press, 1980

Meyer 1990 MEYER, Bertrand: *Objektorientierte Software-Entwicklung* . Hanser ; Prentice-Hall, München u.a., 1990. – ISBN 3-446-15773-5

Meyer 1992 MEYER, Bertrand: Applying "Design by Contract". In: *IEEE Computer* 25 (1992), Nr. 10, S. 40–51

Meyer 1995 MEYER, Bertrand: From Process to Product: Where is Software Headed. In: *IEEE Computer* 28 (1995), August, Nr. 8, S. 23

Meyer 1996 MEYER, Bertrand: Teaching object technology. In: *IEEE Computer* 29 (1996), Nr. 12, S. 117

Miller u. a. 1990 MILLER, Barton P. ; FREDERIKSEN, Lars ; SO, Bryan: An empirical study of the reliability of Unix utilities. In: *Communications of the ACM* 33 (1990), Nr. 12, S. 32–44

Millsap 2010 MILLSAP, Cary: Thinking Clearly about Performance. In: *Queue* 8 (2010), Nr. 9, S. 10–20. – ISSN 1542-7730

Myers 1979 MYERS, G.: *The Art of Software Testing*. J. Wiley, 1979

Naur und Randell 1969 NAUR, Peter (Hrsg.) ; RANDELL, Brian (Hrsg.): *Software Engineering. Report on a conference sponsored by the NATO Science Committee. Garmisch, Germany, 7th to 11th October 1968*. Bruxelles : NATO Scientific Affairs Division, January 1969

Neumann 1987 NEUMANN, Peter G.: Risks to the Public in Computers and Related Systems. In: *ACM Software Engineering Notes* 12 (1987), Nr. 1, S. 3

Neumann 1988 NEUMANN, Peter G.: Risks to the Public in Computers and Related Systems. In: *ACM Software Engineering Notes* 13 (1988), Nr. 4, S. 3

Neumann 1989 NEUMANN, Peter G.: Risks to the Public in Computers and Related Systems. In: *ACM Software Engineering Notes* 14 (1989), Nr. 1, S. 6–9

Neumann 1995a NEUMANN, Peter G.: *Computer-Related Risks*. Addison-Wesley, 1995. – ISBN 0-201-55805-X

Neumann 1995b NEUMANN, Peter G.: Risks to the Public in Computers and Related Systems. In: *Software Engineering Notes* 20 (1995), Nr. 5, S. 8

Neville-Neil 2008a NEVILLE-NEIL, George V.: Code Spelunking Redux. In: *Communications of the ACM* 51 (2008), Nr. 10, S. 36–41

Neville-Neil 2008b NEVILLE-NEIL, George V.: Latency and Livelocks. In: *ACM queue* 6 (2008), Nr. 2, S. 8–10

Neville-Neil 2009 NEVILLE-NEIL, George V.: Kode Reviews 101 – A review of code review do's and don'ts. In: *Communications of the ACM* 52 (2009), Nr. 10, S. 28–29

Norman 1988 NORMAN, Donald A.: *The design of everyday things*. New York : Doubleday, 1988

Ostrand und Weyuker 2002 OSTRAND, Thomas J. ; WEYUKER, Elaine J.: The Distribution of Faults in a Large Industrial Software System. In: FRANKL, Phyllis G. (Hrsg.): *Proc. ACM SIGSOFT 2002 International Symposium on Software Testing and Analysis* Bd. 27(4), ACM, 2002, S. 55–64

Ostrand u. a. 2004 OSTRAND, Thomas J. ; WEYUKER, Elaine J. ; BELL, Robert M.: Where the Bugs Are. In: ROTHERMEL, Gregg (Hrsg.): *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis* Bd. 29, 2004, S. 86–96

O'Sullivan 2009 O'SULLIVAN, Bryan: Making sense of revision-control systems. In: *Commun. ACM* 52 (2009), Nr. 9, S. 56–62. – ISSN 0001-0782

Oz 1994 OZ, Effy: When Professional Standards are Lax: The CONFIRM failure and its Lessons. In: *Communications of the ACM* 37 (1994), Oct., Nr. 10, S. 29–36

Parnas 1972 PARNAS, David L.: On the Criteria to be Used in Decomposing Systems into Modules. In: *Communications of the ACM* 15 (1972), S. 1053–1058

Parnas 1985 PARNAS, David L.: Software Aspects of Strategic Defense Systems. In: *Communications of the ACM* 28 (1985), Nr. 12, S. 1326–1335

Perlis 1982 PERLIS, Alan: Epigrams on Programming. In: *SIGPLAN Notices* 17 (1982), Nr. 9, S. 7–13

Perrow 1999 PERROW, Charles: *Normal Accidents: Living with High-Risk Technologies*. Princeton, N.J. : Princeton University Press, 1999

Petroski 1985 PETROSKI, Henry: *To Engineer is Human: The Role of Failure in Successful Design*. New York : St. Martin's Press, 1985

Raymond 1991 RAYMOND, Eric S.: *The New Hacker's Dictionary*. MIT Press, 1991

Raymond 2003 RAYMOND, Eric S.: *The Art of UNIX Programming*. Addison-Wesley, 2003. – Also available under `http://www.catb.org/~esr/writings/taoup/`

Reenskaug 1979 REENSKAUG, Trygve: Models-Views-Controllers / XEROX PARC. Dec. 1979. – Technical Note. `http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf`

Reiter 2008 REITER, Michael: Softwarepannen wie am Flughafen Heathrow sind auch in Deutschland an der Tagesordnung. In: *Computerzeitung* (2008), 15. Apr.

Rettig 1994 RETTIG, Marc: Prototyping for Tiny Fingers. In: *Communications of the ACM* 37 (1994), Nr. 4, S. 21–27

Reuß 2010 REUSS, Johann: Untersuchungsbericht. In: *Bundesstelle für Flugunfalluntersuchung* (2010), März, Nr. 5X003-0/08. – http://www.bfu-web.de/cln_007/nn_223532/DE/Publikationen/Untersuchungsberichte/2008/Bericht__08__5X003__A320__Hamburg-Seitenwindlandung,templateId=raw,property=publicationFile.pdf/Bericht_08_5X003_A320_Hamburg-Seitenwindlandung.pdf

Rothman 1999 ROTHMAN, Johanna: Retrain Your Code Czar. In: *IEEE Software* 16 (1999), Nr. 2, S. 86–88

Royce 1970 ROYCE, W. W.: Managing the Development of Large Software Systems. In: *Proceedings of IEEE Wescon* IEEE (Veranst.), Aug. 1970

Rödiger und Wilhelm 1996 RÖDIGER, Karl-Heinz ; WILHELM, Rudolf: Zu den Ethischen Leitlinien der Gesellschaft für Informatik. In: *Informatik-Spektrum* 19 (1996), S. 79–86

Schnurer 1988 SCHNURER, K. E.: Programminspektionen — Erfahrungen und Probleme. In: *Informatik-Spektrum* 11 (1988), S. 312–322

Serrano und Ciordia 2004 SERRANO, Nicolás ; CIORDIA, Ismael: Ant: Automating the Process of Building Applications. In: *IEEE Software* 21 (2004), Nr. 6, S. 89–91

Sneed 1988 SNEED, Harry M.: Software-Testen — Stand der Technik. In: *Informatik-Spektrum* 11 (1988), S. 303–311

Sneed und Jungmayr 2011 SNEED, Harry M. ; JUNGMAYR, Stefan: Mehr Wirtschaftlichkeit durch Value-Driven Testing. In: *Informatik-Spektrum* 34 (2011), Nr. 2, S. 192–209

Sodan 1998 SODAN, A. C.: Yin and Yang in Computer Science. In: *Communications of the ACM* 41 (1998), Nr. 4, S. 103–111

Spolski 2010 SPOLSKI, Joel: *Hg Init: a Mercurial tutorial*. http://hginit.com. 2010

Störrle 2005 STÖRRLE, Harald: *UML 2 für Studenten*. Pearson Studium, 2005

Swartz 1996 SWARTZ, A. J.: Airport 95: Automated Baggage System? In: *Software Engineering Notes* 21 (1996), Nr. 2, S. 79–83

Süddeutsche Zeitung 2010 SÜDDEUTSCHE ZEITUNG: Der Bordcomputer ist schuld. In: *Süddeutsche Zeitung* (2010), 5. März, Nr. 53, S. 9

Traufetter 2010 TRAUFETTER, Gerald: Ermittler werfen Airbus mangelhafte Piloten-Information vor. In: *Spiegel online* (2010). – `http://www.spiegel.de/wissenschaft/technik/0,1518,681655,00.html`, zuletzt gesehen 9. März 2010

Valk 1997 VALK, Rüdiger: Die Informatik zwischen Formal- und Humanwissenschaften. In: *Informatik-Spektrum* 20 (1997), Nr. 2, S. 95–100

Voas 1997 VOAS, Jeffrey: How Assertions Can Increase Test Effectiveness. In: *IEEE Software* 14 (1997), Nr. 2, S. 118–119,122

Voas u. a. 1997 VOAS, Jeffrey ; MCGRAW, Gary ; KASSAB, Lora ; VOAS, Larry: A 'Crystal Ball' for Software Liability. In: *IEEE Computer* 30 (1997), Nr. 6, S. 29–36

Weiss 2008 WEISS, Harald: CAD-Inkompatibilität verursacht bei Airbus Milliardenschäden. In: *Computerzeitung* (2008), April, Nr. 15

Wikipedia 2008 WIKIPEDIA: *Toll Collect*. `http://de.wikipedia.org/wiki/Toll_Collect`. Jan. 2008

Wirth 1978 WIRTH, Niklaus: *Systematisches Programmieren*. Teubner, 1978

Wirth 1995 WIRTH, Niklaus: A Plea for Lean Software. In: *IEEE Computer* 28 (1995), Nr. 2, S. 64–68

Zemanek 1979 ZEMANEK, Heinz: Abstract Architecture—General Concepts for Systems Design. In: BJØRNER, Dines (Hrsg.): *Abstract Software Specification—Proceedings of 1979 Copenhagen Winter School*. Springer, 1979 (Lecture Notes in Computer Science 86), S. 1–42